TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JUUSO TAPANINEN

SCALABILITY OF A NODE.JS BASED MICROSERVICE ARCHITECTURE ON HEROKU

Master of Science thesis

Examiner: Prof. Kari Systä
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 5th May 2018

ABSTRACT

JUUSO TAPANINEN: Scalability of a Node.js based microservice architecture on Heroku
Tampere University of Technology
Master of Science thesis, 49 pages, 4 Appendix pages
June 2018
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Kari Systä
Keywords: microservice, API gateway, Platform as a Service, Node.js

Microservices are a method for creating distributed services. Instead of monolithic applications, where all of the functionality runs inside the same process, every microservice specializes in a specific task. This allows for more fine-grained scaling and utilization of the individual services, while also making the microservices easier to reason about.

Push notifications can cause unexpectedly high loads for services, especially when they are being sent to all users. With enough users, this load can become overwhelming. Most services have three options to meet this increased demand: scale the service either horizontally or vertically, improve the performance of the service or send the notifications in batches. In our service, we chose to implement the batched sending of notifications. This caused issues in the amount of time it took to send all notifications. Instead of a short peak in traffic, the service had to manage consistently high load for a long period of time.

This thesis is part literary study, where we research microservices in more detail and go through the more common architectural patterns associated with them. We explore a production service that had issues with meeting the demand during high load caused by push notifications. To understand the production environment and its restrictions, we also explain the runtime, Node.js, and the cloud provider, Heroku, that were used. We go through the clustering implementation details that allowed our API gateway to scale vertically more effectively.

Based on our performance evaluation of an example Node.js application and our production environment, clustering is an easy and effective way to enable vertical scaling for Node.js applications. However, even with better hardware, there still exists a breaking point where the service can not manage any more traffic and autoscaling is not good enough to meet the demand. A service requires constant monitoring and performance improvements from the development team to be able to meet high demand.

TIIVISTELMÄ

JUUSO TAPANINEN: Node.js pohjaisen mikropalveluarkkitehtuurin skaalautuvuus Herokussa
Tampereen teknillinen yliopisto
Diplomityö, 49 sivua, 4 liitesivua
Kesäkuu 2018
Tietotekniikan koulutusohjelma
Pääaine: Pervasive Systems
Tarkastajat: Prof. Kari Systä
Avainsanat: mikropalvelu, API-yhdyskäytävä, ulkoistettu palvelualusta, Node.js

Mikropalvelut ovat tapa tehdä hajautettuja järjestelmiä. Monoliittisissa sovelluksissa kaikki toiminnallisuus ajetaan samassa prosessissa, kun taas mikropalveluissa jokainen palvelu keskittyy yhteen tiettyyn toiminnallisuuteen. Tämä tekee mikropalveluista helpommin ymmärrettäviä.

Push-viestit voivat aiheuttaa odottamattoman paljon liikennettä palveluille, erityisesti kun niitä lähetetään kaikille käyttäjille. Useimmilla palveluilla on kolme vaihtoehtoa tämän kasvavan liikenteen käsittelyyn: palvelua voidaan ajaa useammilla tai tehokkaammilla palvelimilla, palvelun tehokkuutta voidaan parantaa tai push-viestit voidaan lähettää erissä. Me valitsimme push-viestien lähetyksen erissä ratkaisuksi. Tämä johti pitkään läpivientiaikaan viestien lähetyksille. Yksittäisen korkean liikennepiikin sijaan palvelun piti pystyä vastaamaan pitkäkestoiseen ja korkeaan kuormaan.

Tämä työ on osittain kirjallisuustutkielma, joka keskittyy mikropalveluihin ja niiden yleisiin arkkitehtuurivalintoihin. Tutustumme meidän tuotantoympäristöön ja siellä push-viestien aiheuttamaan ongelmaan korkean kuorman käsittelyssä. Tuotantoympäristömme ja sen rajoitteiden ymmärtämistä varten käymme läpi myös meidän ajonaikaisen ympäristön eli Node.js:n ja Herokun. Käymme läpi klusterointiin liittyvät muutokset, jotka mahdollistivat tehokkaampien palvelimien hyödyntämisen meidän API-yhdyskäytävässä.

Node.js pohjaiseen esimerkkisovellukseen tehtyjen kuormitustestien ja tuotantoympäristömme tulosten perusteella klusterointi on helppo ja tehokas tapa mahdollistaa Node.js pohjaisen sovelluksen käyttö tehokkaammilla palvelimilla. Tehokkaammilla palvelimillakin tulee kuitenkin jossain vaiheessa vastaan niin korkea kuorma, että palvelimet ja automatisoidut järjestelmät kapasiteetin lisäämiselle eivät enää kykene siihen vastaamaan. Palvelu vaatii jatkuvaa kehitystä ja tarkkailua kehittäjätiimiltään, jotta se voi vastata kasvavaan kysyntään.

PREFACE

I have thoroughly enjoyed the years I spent at Tampere University of Technology. I have learned and lived a lot during my time at the university. I would like to thank the Guild of Information Technology for making my stay all the more enjoyable.

I would like to thank Futurice, and especially Jussi Riihelä for his continued support during the writing process of this thesis. I am sure this thesis would still be a work in progress if it were not for our regularly scheduled meetings discussing my progress. I would also like to thank my thesis supervisor Kari Systä for his continued insight and guidance throughout the whole process.

I would like to thank my family for their support in general and during my studies.

Tampere, May 23, 2018

Juuso Tapaninen

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application programming interface
AWS	Amazon web services
CAP	Consistency, Availability, Partition tolerance
CLI	Command Line Interface
DNS	Domain Name System
DSL	Domain-specific language
FIFO	First In, First Out
HATEOAS	Hypertext as the engine of application state
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
I/O	Input/Output
IP	Internet Protocol
IPC	Inter-process communication
JSON	JavaScript Object Notation
JVM	Java virtual machine
LAN	Local Area Network
PaaS	Platform as a Service
RCP	Remote procedure call
REST	REpresentational State Transfer
SaaS	Software as a Service
SCM	Source Control Management
SOA	Service oriented architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VCS	Version Control System
XML	Extensible Markup Language

1. INTRODUCTION

Traditionally, procuring hardware has been a big undertaking for companies. Ordering, setting up and configuring server hardware is a process that can easily consume a month of calendar time, if not more, since server hardware is rarely available off the shelf, due to the multitude of different configuration options [1, p. 85]. Hardware is also a large financial investment, one which smaller companies would often rather live without. Mistakes in capacity planning can also lead to catastrophic failures once the product launches

Self-hosted hardware comes with a heavy maintenance burden. Being responsible for the whole hardware and software stack is most likely not the core competency of most companies.

Modern cloud hosting providers like Amazon Web Services (AWS) and Google Cloud Platform (GCP) offer more elasticity when it comes to hosting options. Infrastructure as a Service (IaaS) [2, p. 13-15], Platform as a Service (PaaS) [2, p. 15-17] and Software as a Service (SaaS) [2, p. 17-18] allow different options of hosting where maintenance gets shared between the service provider and the client to varying degrees. Hardware is always taken care of by the service provider, and due to modern virtualization techniques, adding new instances and scaling up happens in minutes instead of months. Ability to utilize different cloud based resources effectively is seen as a business advantage. Since cloud based resources are often priced by the hour or based on usage, even smaller companies and startups can start with smaller investments or test different configurations for their applications very cheaply [2, p. 2]. Even some larger companies like Netflix have moved their infrastructure completely to the cloud [3] due to the amount of elasticity and freedom it provides: scaling servers up and down based on usage can lead to cost savings. Pay as you go payment models allow for quick and cheap experimentation while trying to mix and match the best possible hardware and software for the problem at hand.

Typically, web applications are ran as a singular process, scaled to multiple different servers when or if necessary. This means that if a specific part of the service receives a disproportionate amount of the overall traffic, our only option is to scale the whole service to meet the necessary demand [4]. Easily available computing resources have also led to alternative ways of building software, that allow us to utilize modern cloud based

hosting solutions more effectively.

Microservices split the functionality of a product into multiple independent services that communicate with each other over the network through an application programming interface (API). Since the services are independent, they can also be scaled independently based on demand. Smaller services can also be easier to reason about, since they tend to focus on a single piece of functionality.

Both the hosting environment and the runtime of a service can introduce restrictions on how a service is run. Modern PaaS environments often have restrictions in place for different aspects of running an application on the platform. For example, third party applications like databases or caches are managed by the platform and can not be run inside the application servers. Runtimes like Node.js can also impose their own set of restrictions on the application. On Node.js, all the user created code runs inside a single thread and execution happens asynchronously.

API gateways are a common approach to make a fleet of microservices seem like a single service to the API consumer. An API gateway consolidates all of the individual microservice APIs and offers them as a cohesive set of APIs to the consumers. This makes it seem like there is a single service handling all of the requests on the API caller side, while the reality is more complex. The API gateway approach does have a negative effect of forcing all of the traffic to route through the same service, which means that the gateway has to be able to handle the combined traffic from all of the individual microservices.

In this thesis, we focus on finding the answers to the following research questions:

1. What are microservices and how to build and manage them?
2. What limitations does our environment impose on our services and scalability?
3. How did we fix our API gateway scalability issue?

This thesis is organized as follows: in Chapter 2 we go through the basics of microservices in more detail. Chapter 3 explores our production service runtime, Node.js. Chapter 4 introduces Heroku, a Platform as a Service hosting provider that our production service uses. Chapter 5 explains how push notifications caused major traffic spikes on our services. Chapter 6 goes through the implementation details of the changes we made to our API gateway. Chapter 7 explores the performance impact of Node.js clustering. Chapter 8 includes conclusions and future ideas for improving our API gateway performance.

2. MICROSERVICES

Microservices are small, autonomous, and distributed systems that work together [5, p. 22]. These small, individual systems are modeled after concrete business needs or functions, where every service is responsible for a specific portion of a larger system. As a whole, these services are utilized to achieve a plethora of different business goals, perhaps even ones that were not originally thought of.

Microservices can be seen as an evolution of service oriented architecture (SOA), sometimes it is even called rebranded SOA [6, p. 1]. Service oriented architecture is a design approach, where multiple individual services collaborate to provide a cohesive set of capabilities [5, p. 8]. These services communicate across the network, rather than method calls within a single process.

”SOA at its heart is a very sensible idea. However, despite many efforts, there is a lack of good consensus on how to do SOA *well*.” [5, p. 8] The main difference between microservices and service oriented architecture is that microservices lack the ambiguity of SOA. There is no clear definition for what SOA actually is: to some it means allowing systems to communicate with some standard structure like XML, to others it is about asynchronous messaging between services.

Microservices are the opposite of monolithic services, where all of the functionality runs inside a single process. In a monolithic service, implementation can be divided by modules, classes or some other mechanism provided by the programming language being used. These modules may have well defined application programming interface (API) boundaries between each other, but they all live inside the same code base.

2.1 How microservices are built

A clear definition for how microservices should be built does not exist. Some organizations like to enforce rules related to size of a service, which is actually one of the downsides of the term microservices. Services should be modeled based on business needs and while smaller services are preferable, they are not the goal.

One common approach for creating microservices is starting with an existing, monolithic application, and using microservices as a refactoring pattern. Because understanding of the business domain is most likely limited at the start of a project [5, p. 33], creating new microservices can be problematic. Refactoring the existing, stable APIs of a monolith into new services is an easier way to get acquainted with microservices.

2.1.1 Break the monolith

Breaking down an existing service in to smaller chunks starts with a thorough examination. Hopefully the service utilizes a clearly defined module structure with good APIs between the different modules. Starting with smaller modules makes it easier to get familiar with implementing microservices.

Teams can also focus on creating microservices from new features that are being created and ones that are on the horizon. This way the existing monolith does not have to be changed too much at the start. Old functionality can instead be slowly transitioned to the appropriate new services that have been created while making new features. [6, p. 65]

Incremental transition is the safest way to proceed when changing an existing service. Trying to break all of the modules in to microservices in a single “Big Bang” sort of approach is most likely to end up failing.

2.1.2 Communication between services

Since microservices are independent, distributed services, they need to communicate between each other through the network. There are multiple different ways to implement this communication, and some services may even choose to utilize multiple different methods at the same time.

Communication between service boundaries can be either synchronous or asynchronous. An example of synchronous communication can be the request response model: a client sends a requests and waits until the server responds with the result. A message queue on the other hand is a good example of asynchronous communication: clients add work to the queue and does not wait for it to be completed. The queue distributes the work to the relevant workers.

Synchronous communication

Remote procedure calls (RPC) is a technique where a local function call is actually executed on a remote server [5, p. 46]. There are multiple different implementations of RPC, like Java RMI [7] or Thrift [8]. Many RPC implementations are binary in nature, like Thrift, but some use textual representations, like how Extensible Markup Language (XML) is used with Simple Object Access Protocol (SOAP). There are many different networking protocols in use, like the Hypertext Transport Protocol (HTTP) as well as Transmission Control Protocol (TCP) or User Datagram Protocol (UDP).

RPC tries to hide remote calls as local function calls, which can lead to issues. Assuming that the network is always reliable is one of the eight fallacies related to distributed computing [9]. This is why clear distinction between local and remote calls is always preferable.

Some RPC implementations can also be very brittle. Java RMI can generate stubs for clients and servers, but it also requires the client and server to have matching object descriptions. This forces client and server releases to happen at the same time, which makes us lose out on one of the key benefits of microservices, independent deployments.

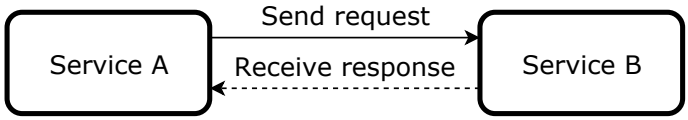


Figure 2.1 *Synchronous communication. Service A calls B directly and waits for the response until proceeding. This call is done over the network and is subject to connection issues and timeouts that are not a problem when calling functions inside the same process*

Figure 2.1 is an example of synchronous communication. When two services are communicating synchronously, the calling service waits until the service receiving the call responds. Since the requests are done over the network, they are inherently more error prone than function calls inside the same process.

REpresentational State Transfer (REST) is a web inspired architectural style. In REST, the concept of resources is key. The server might have multiple different resources that the client can ask different representations for. The client can ask for a resource in JavaScript Object Notation (JSON) or XML and the server responds as requested, based on its capabilities.

REST is built on top of HTTP, and it utilizes multiple aspects of it. HTTP verbs like GET, POST and PUT are used to denote fetching a resource, creating a new resource and updating an existing resource [5, p. 50]. Utilizing HTTP gives us a lot of existing tooling

like caching, load balancing and monitoring utilities.

REST also introduces the concept of hypermedia as the engine of application state (HATEOAS). The idea of HATEOAS is that REST resources can contain links between other resources in a lot of different formats. The clients should perform their operations by utilizing and navigating these links. [10]

Clients utilizing HATEOAS do not need to know the exact location of all the available REST resources. They can navigate the links between resources to find the necessary information. Unfortunately, HATEOAS results in APIs that require a lot of back and forth communication between the server and the client, which is why it is pretty seldom utilized inside RESTful services. [5, p. 52]

The main difference between REST and RPC is that REST is just a way to build HTTP APIs. Calling a RESTful service requires HTTP communication with another server, so it is more explicitly a request going over the network than an RPC function call is in many cases.

REST is a common choice for service to service communication, but it has downsides. Utilizing HTTP for communication is most likely not suitable for low latency needs. REST also does not generally have the stub generation features that some RPC implementations may have.

Asynchronous communication

Asynchronous and event based communication can be achieved by utilizing message brokers. There are multiple message broker implementations available, but a good, free example of one is RabbitMQ [11]. In an event based message broker pattern there are producers and consumers.

RabbitMQ can be configured in two different ways: work queues and publish/subscribe. In work queue mode, producers add messages to a queue. Work is divided between all the registered consumers and every message is processed at least once. This workflow is shown in figure 2.2.

In publish/subscribe mode, instead of a queue, producers send messages to an exchange. The exchange is responsible for pushing messages to queues based on the exchange type. Typically in a publish/subscribe mode, the exchange is configured to relay all messages to every consumer. This means that every consumer has a queue of its own and the exchange adds the messages to every queue it is aware of.

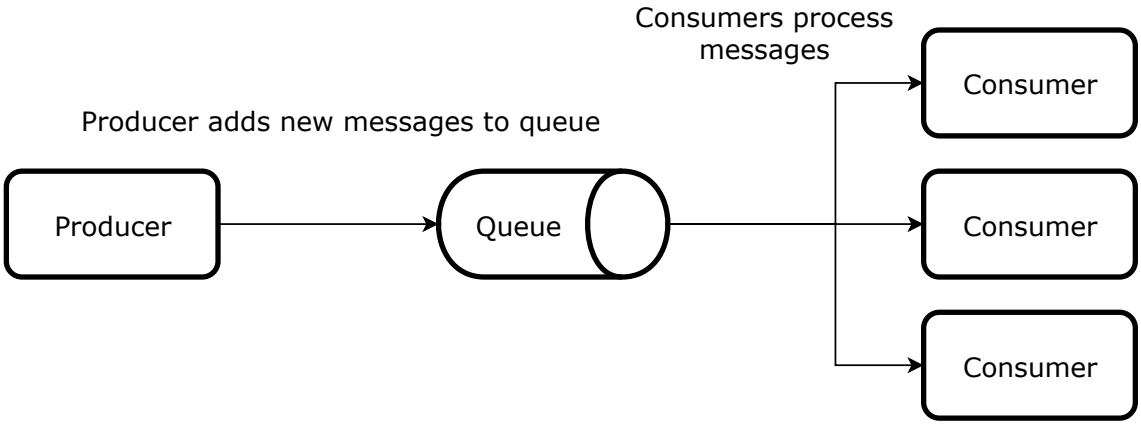


Figure 2.2 *Asynchronous communication. Producers know of the queue, but are not necessarily aware of the different consumers for their messages. This enables better decoupling between services*

This sort of asynchronous, de-coupled form of communication allows producers to not know what systems are actually going to react to the events that they produce, making it easier to add new or additional steps to the event chains. Message broker patterns are built to be resilient and scalable, but this scalability does not come for free [5, p. 55].

The biggest problem with asynchronous communication is the added complexity. Besides the fact that services need to add additional code for producers and consumers to be able to function, there is the additional complexity of the message broker itself. This is an additional process or service that needs to be managed along other services.

When the producers do not actually know all of the consumers, the best one can hope for in terms of data consistency is eventuality: given enough time, the system and data will end up in a consistent state. This is due to the CAP theorem, which states that a distributed systems can, at best, have two of the three following guarantees: consistency, availability and partition tolerance [12]. Since distributed transactions are non-feasible almost all of the time, most services choose availability and partition tolerance in favor of consistency, settling for eventual consistency. The CAP theorem is true for all distributed services, not just asynchronously communicating ones, but it is a big consideration when utilizing asynchronous communication patterns.

2.1.3 Service discovery and service composition

In a distributed environment with services communicating over the network, Internet Protocol (IP) addresses become the basic building blocks of those services: IP addresses are used to identify and communicate between services. Hard coding or configuring these values becomes unwieldy with a large amount of services and may even be an ill-advised

venture in some hosting environments: IaaS and PaaS providers often give servers or computing units dynamic IP addresses that can change on every reboot.

The Domain Name System (DNS) can be seen as a lightweight form of service discovery: instead of using IP addresses, we utilize domain names that eventually resolve in to IP addresses. Services do not often talk to one another directly. Instead, load balancers are utilized between service boundaries, which handle redirecting traffic to the actual application servers. When DNS is used for service discovery, we can setup the domain names to resolve to the load balancers, which will route the traffic to the actual services. Most IaaS or PaaS service providers will provide a method for automatic traffic routing from the load balancer to the server instances.

A load balancer will either provide a static IP address for the domain, or another method that will make it discoverable from the same address at all times. The downsides of utilizing DNS as a method of service discovery are mainly the amount of configuration and maintenance required. Every service requires a domain name to be registered. Changes on the service level are minimal, however, since instead of IP addresses, services can just use the domain names to communicate between one another.

Another alternative for service discovery is utilizing a service broker. A service broker is basically a highly available, distributed key-value storage, which has information on all of the services that are running and their locations. An example of a distributed, highly available service broker is Consul [13] by HashiCorp. Every service knows the location of the service broker and it is the job of the service broker to know the location of every other service.

In a service broker configuration, every service that comes online registers themselves and their address with the service broker. The service broker can also query the health of the registered services. This way the service broker can avoid directing traffic to unavailable nodes. Other services can query for the locations of the services that they need and they can continue using the received location information until the address becomes unresponsive, which is when they will query for the potential new address again from the service broker.

Main issue with the service broker method of service discovery is the additional logic required at the service level: registering the service and querying the service broker for other service addresses, caching the responses and re-querying for a new address on connection failure. The service broker itself also has to be highly available to ensure that communication between services does not fail just because the service broker was down. The service broker can be seen as a single point of failure, if it happens to go down.

2.1.4 Gateway

While API consumers can access a microservice architecture based service by using the same service discovery pattern that a service might use internally, this requires a lot of complexity from the service consumer. Instead of a single API, consumers need to manage multiple different APIs and possibly varying authentication schemes. This is what the gateway pattern aims to simplify.

In the gateway pattern, external consumers have a single API they can call that proxies requests further to the relevant microservices. This makes it easier to unify authentication schemes required for external API consumers and hides the internal complexity of the service from the consumers [6, p. 15]. Due to the popularity of this pattern, there are also existing solutions out there, like the AWS API Gateway [14] or Kong [15].

The gateway pattern can also help in high latency applications like mobile networks, where doing multiple simultaneous and separate API calls just to achieve one thing is costly. Results from multiple actual services can be combined behind a single API endpoint at the gateway level, which again simplifies the usage of the service on the consuming side.

The problem with the gateway approach is mainly that it needs to be efficient at managing the incoming and outgoing traffic. It reroutes a lot of traffic, so bad performance on the gateway level will make service worse for all API consumers, and may lead to service outages if the gateway goes down. This is why the gateway should be highly available and it should preferably do minimal additional processing besides the traffic routing itself.

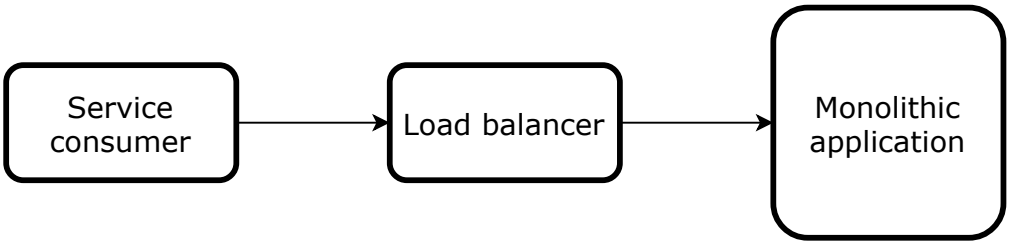
2.2 Scalability

There are two different ways to scale services: horizontally and vertically. Horizontal scaling means adding more individual machines to distribute the load. Vertical scaling means upgrading existing hardware. [1, p. 155-157]

Figure 2.3 represents what horizontal and vertical scaling means for a monolithic service. Both figures have a load balancer between the service and the service consumer. Load balancers are responsible for distributing load between multiple servers. In the horizontal scaling example 2.3a, the service can be scaled by adding additional servers behind the load balancer. The larger box in the vertical scaling figure 2.3b represents a larger server instance being used.

In a monolithic service, everything is running inside the same process. This makes it impossible to scale individual parts of the same service. For example, if the search functionality receives two times the traffic of another part of the service, we can not just scale

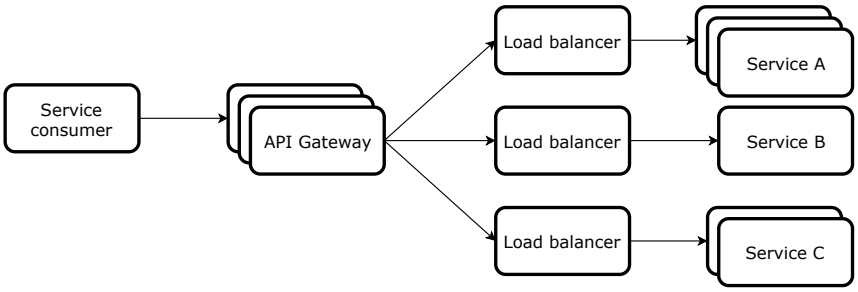
(a) Horizontal scaling



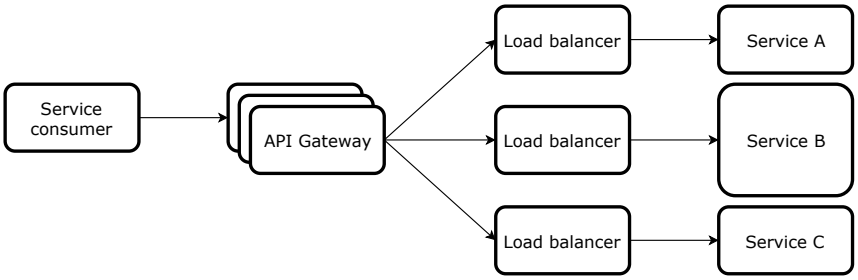
(b) Vertical scaling

Figure 2.3 *Horizontal and vertical scaling for monolithic applications*

the search. We need to scale everything along with it. [4]



(a) Horizontal scaling



(b) Vertical scaling

Figure 2.4 *Horizontal and vertical scaling for microservices*

Figure 2.4 demonstrates how scaling works with microservices. In these examples the API gateway pattern is utilized. Both the horizontal 2.4a and vertical 2.4b scaling examples show that microservices can be scaled individually.

Being able to scale services individually allows for more flexibility. In a monolithic application, it makes most sense to use instance types optimized for generic computing, since it is not possible to run parts of the same process on different hardware. With microservices, instance selections can be optimized on the individual microservice level.

2.3 Benefits of utilizing microservices

Microservices are not a silver bullet, but there are situations where they do excel. Properly managing a fleet of microservices requires a lot from a team, but depending on team composition and business needs, they can be the right choice as the architecture pattern for a problem.

2.3.1 Smaller services are easier to reason about

Monolithic services, especially ones that have been developed for multiple years, can grow quite large in terms of the amount of functionality, code, and tests. Over time, the clear module boundaries of the service can also start to corrode and eventually similar functionality starts to spread across the whole service [5, p. 2]. This can make the service quite difficult to reason about.

With microservices, every service should solve a single and concise business need. For example, lets say that we need to store and manage loyalty points for our customers based on their purchases. The microservice created for solving this need should only handle tasks related to loyalty points: creating a new balance for a registered user, updating the balance and so on. Even the idea of an user might be foreign to our loyalty service: we only have an identifier pointing to the user. User details can be managed by another microservice.

When services are split into these smaller units, each service has a clear focus on a specific implementation detail. The developers know where all of the functionality related to this single piece of the whole ecosystem is located at, and the service will most likely end up being more simple due to having less code and functionality associated with it. [5, p. 2]

2.3.2 Testing

Testing a monolith can become a hassle. Just like the code itself, tests can become a mess or the functionality itself might be hard to create tests for since everything is so interconnected with everything else. Creating testable code requires focus and discipline, and retrofitting tests might be impossible

With microservices, testing becomes a bit more simplified, at least when compared to the monolithic approach. Data that the microservice itself is not responsible for will be provided by other services via remote calls, which can be easily stubbed or faked in testing. Testing does not become trivial just because the services are smaller, and testing the fleet of microservices as a whole will require additional effort, but on an individual service level tests do become simpler, easier to reason about.

2.3.3 Right tools for the job

Microservices allow development teams to choose the tools that best suit them and the needs of the service. With monolithic applications, both large and small technology constraints are formed at the start of a project. Large technology constraints like a programming language, can be hard or even impossible to change later on. Smaller technology constraints, like libraries or coding patterns, can usually be swapped, but may require some effort to do so. Microservices can be built with a right tool for the job mindset.

Since services talk to each other over the network through an API, the actual implementation details of the service are not important to API consumers. This means that inside a single microservice ecosystem, services can be built with multiple different languages, varying tooling, and data stores [16, p. 2]. Development teams can make autonomous decisions about what is best for their service, as long as the API consumers can get their needs fulfilled by the actual implementation.

While development teams can choose their tools freely, it is sensible to have some defaults. For example, Netflix has chosen the Java virtual machine (JVM) as their default platform and has developed most of their tooling on top of it [17]. Teams are free to choose something else, but they will lack the internal support that Netflix has for the JVM, if they decide to go with an alternative approach.

2.3.4 Resilient services

Microservices can enable development teams to have more resilient services. In his book Building microservices, Sam Newman describes how an isolated failure of a single microservice does not necessarily bring down the whole system, if built properly [5, p. 5]. When a monolith goes down, every part of the service stops working with it. A microservice based solution can still function at a limited capacity, if parts of the distributed system are down.

For example, an e-commerce application could still show products and offers, even if the actual order service was momentarily down. Unfortunately, this resiliency does not come without a cost and teams can easily build services that may be susceptible to cascading failures. A cascading failure is an error state, where one service going down takes other services down with it.

2.3.5 Deployment

Deployments with larger services can be painful, which is why teams often accrue a lot of smaller changes and delay deployment until absolutely necessary. Unfortunately, releasing a lot of smaller changes at once can also increase the likeliness of failure. This is why developers should be able to release early and release often: releases should be so easy to do that they can be done multiple times a day if necessary [1, p. 326-327].

Since microservices are smaller, more isolated pieces of functionality, teams are able to deploy changes to a single service without affecting the rest of the system [6, p. 9]. Botched deploys are easier to notice and faulty code can be rolled back to an earlier state on a service by service level. This allows teams to iterate faster: fixing a simple bug can be deployed to production in minutes to hours instead of days to weeks.

2.3.6 Finding new uses for existing functionality

Monolithic services are often built with a specific use case in mind. Depending on the actual implementation details, this can really narrow down the ability to re-use the implemented functionality for something else later on.

Since microservices are autonomous by definition, having an application programming interface (API) is a requirement instead of a nice to have feature for the service. Having well defined APIs for services built around core business needs can enable companies to compose the available functionality in new and different ways. Multiple different API consumers like websites and mobile apps are a given, of course, but APIs can even bring up combinations and use cases for the data, that were not originally thought of.

Data is the most valuable asset for a lot of companies. Making it readily available in an easy to consume ways can enable new and more agile ways of working and allow for more rapid evolution of businesses.

2.4 Downsides of microservices

While microservices can be a beneficial approach to some situations and team compositions, they are not without their downsides and additional complexities.

2.4.1 Fallacies of distributed computing

Because microservices are distributed, they are susceptible to the fallacies of distributed computing [9]. These fallacies are common errors or false assumptions that developers

often make when developing distributed services. The following paragraphs explain the different fallacies in more detail.

The first fallacy claims that **the network is reliable**. This is a false assumption since both networking hardware and software can fail. Given enough volume of traffic, failures in the network are inevitable [18]. Networks can also face physical issues like cables being disconnected or power outages.

Assuming that **latency is zero** is also a fallacy. While latency can be quite good on a Local Area Network (LAN), once we cross over to Wide Area Network (WAN), latency is something that needs to be taken into account. Assuming zero latency over a network can make one assume that networked calls are like local calls, but this assumption becomes even more false when the other fallacies are taken into consideration. [18].

Bandwidth being infinite is another common assumption that is not true. While physical network connections are fast, the amount of data and the speed with which we would like to process that data has also progressed. Networks can also be affected with packet loss, which diminishes the amount of bandwidth available [18].

Thinking that **the network is secure** is also bad form. Even legitimate users can unknowingly spread malware, and there are a lot of bad actors actively trying to break services [18]. Security should be built in from the start and potential risks should be evaluated. Security is always a tradeoff of costs, risks and the probability of those risks occurring.

Topology not changing is false both on the client and the server side. A service provider might move applications to other servers at a moments notice, which changes IP addresses and makes them unreliable [18]. Clients, especially mobile ones, can change between different forms of wireless and wired networks constantly.

Having **one administrator** is an unrealistic expectation both on the service provider side, as well as the application development side, especially when microservices are concerned. Every service might have a different team responsible for it. Service providers have multiple administrators responsible for their massive amounts of infrastructure [18].

Transport cost is never zero in a distributed environment. There is a clear cost associated with bandwidth in services like AWS or Google Cloud, where it is often billed by the gigabyte. Serializing data for transfer is another important cost related to data transportation [18].

While assuming a **homogenous network** is not a big issue on the lower levels of the networking stack, it is unlikely to be true for network clients and the applications running on the network. There are multiple different operating systems like Linux, Windows,

Android and iOS likely running on the same network [18]. Proprietary protocols should be avoided to make interoperability between different clients as easy as possible.

When communicating over the network, developers should always assume the worst. Services being unavailable, unresponsive or slow can be common things that we should account for. When a singular service is slowing down, it can lead to a cascading failure in other parts of the system: requests start to queue up and consume more memory than the servers can handle, until they go down. Timeouts can help with slow services, but they do not completely remove cascading failures from the equation. If the timeout is too long and the requests are coming in too fast, a slow service can still lead to cascading failures.

The circuit breaker pattern [19] can be used to help with potential cascading failures. A circuit breaker utilizes timeouts and health checks to determine the overall health of the remote service. Once a circuit breaker is tripped, the remote calls will fail instantly for a certain time. This way requests will not start queueing up due to an unresponsive server [20, p. 9]. The circuit breaker can slowly start to ramp up the traffic going through when the remote service recovers. Failing instantly for a portion of API calls is better than failing to serve anything due to the servers going down.

2.4.2 CAP theorem

The CAP theorem states that in a distributed system, only two of the following three guarantees can be true at the same time: consistency, availability, and partition tolerance. Since microservices are distributed, the CAP theorem applies.

Consistency is something that users expect from services and it is what we strive for when choosing Atomicity, Consistency, Isolation, Durability (ACID) compliant databases [12]. Services are expected to function in a transactional manner: changes should be only committed if everything succeeds. Otherwise the whole transaction should be rolled back.

Web services are expected to be **highly available**. Sent requests should all receive a response [12]. Services being down can have unexpected results, and often the services are needed the most when they are unavailable.

A distributed network should be **tolerant towards partitioning**. If a single node of a service crashes, the service should still be able to recover [12].

Since distributed transactions are hard to do and can be costly to run, most services choose availability and partition tolerance. At best, distributed systems can settle for eventual consistency. This means that the data for the service might be inconsistent at a single point in time, but given enough time it will end up in a consistent state [12]. Being able

to achieve this requires some effort from the developers. All the possible error situations that can crop up when syncing state between service boundaries have to be handled.

2.4.3 Performance

While performance on the individual service level can be on par with monoliths, whenever we have to go over the network for an answer, there is an inherent performance hit over a local function call, like inside a monolith. The actual impact on performance depends on the way the services and their data has been separated from one another, but unnecessary service boundary communication should be avoided, if at all possible.

2.4.4 Overextending a team

More services to maintain leads to more operational complexity for the team in charge of maintenance. A small team in charge of a fleet of microservices can lead to issues in both maintenance and monitoring.

Monitoring tasks become more complicated with multiple services [5, p. 11]. When service boundaries are crossed, details on which request was related to what can become fuzzy. This can be avoided with good logging and attaching some sort of identifiers to requests, but requires effort to accomplish.

In a monolith, dependencies and their updates are maintained for a single service only, and everything is contained in a single repository. With microservices, there is additional complexity in maintenance, since every service needs to be managed individually. If different teams are responsible for different services, there can be both large and small variances between microservices. This makes moving between projects and tasks more complicated, since there is more of a context switch involved in moving on to another service.

2.4.5 Microservices necessitate automation

Traditional production deployments can sometimes be really complicated, involving a lot of manual steps and man-hours to complete [1, p. 326-327]. While these can be manageable with a single service, there is no room for manual procedures in the microservice world. When there are multiple different services to manage, everything that can be automated, should be.

Microservices require a lot from a team when it comes to workflows and automating them [20, p. 40]. Continuous deployment, continuous integration and automated deployments

become almost a necessity to be able to successfully manage all the complexity of multiple individual services.

3. JAVASCRIPT ON THE SERVER WITH NODE.JS

JavaScript is a dynamically typed scripting language. It was originally created by Netscape Communication for their Netscape Navigator browser [21], but was later standardized by Ecma International as ECMAScript. Modern browsers all support the ECMAScript standard, to varying degrees. JavaScript is an implementation of the ECMAScript standard, and as such, it can provide extensions not included in the standard. These extensions may differ from one browser manufacturer to another.

Node.js is a JavaScript runtime designed for server side execution [22]. It is based on the V8 JavaScript engine created by Google [23], which was originally made for JavaScript execution on the Google Chrome web browser. Originally released in 2009, it has since gathered a massive following and an ecosystem to match.

Before Node.js, JavaScript programmers were mainly constrained to use the web browser as their platform of choice. The release of Node lead to a lot of web frontend developers being able to develop on the server side for the first time. Having a familiar language with familiar conventions on the server has been one of the larger reasons for Node.js adoption as a server side technology.

Multithreading allows programs to better utilize the available resources in a multi-core system, or allows a single core system to change execution context from one thread to another while we are waiting for input/output (I/O) to finish. A web server could have multiple execution threads, each ready to serve a single incoming query, which would allow the service to serve a lot more requests than a non-multithreaded version due to all of the synchronous execution calls slowing down the request handling. [24]

Multithreading is widely used successfully, but it does not mean that it is easy. Threads can become deadlocked, meaning that they can not continue their execution unless another thread releases a lock or does something else, that then never happens. Sharing data or state between threads can result in some really hard to figure out issues that can be complicated to reproduce.

3.1 Asynchronous execution

Typically JavaScript execution is constrained to a single thread or process, and the same constraint is true for Node.js. Node.js handles concurrency by utilizing the event-driven asynchronous programming model. This is something that is familiar from traditional web user interface programming, where we can register event handlers to various events like button clicks or page loads.

In an asynchronous execution model, everything I/O constrained is delegated to be handled through events and their associated callbacks or event handlers. This allows the code execution to jump from place to place while we are waiting on I/O bound things to finish so that we can continue processing the results. While this is not true concurrency like multithreading in a multi-core system, I/O tasks like networking or data handling on disk are often the parts of our programs where most of the time is spent. Instead of blocking and waiting for these operations to finish, execution can move on to other things while waiting for the relevant events to fire.

Since a lot of operations in a typical server environment are I/O bound, asynchronous execution is a good fit for server side programs. Waiting for database queries to return, disk reads or writes to finish or external service requests to come back. While these events are waiting to happen, code execution can move on to things that have already returned.

```
1  const sys = require('sys')
2  const http = require('http')
3  const url = require('url')
4  const path = require('path')
5  const fs = require('fs');
6
7  http.createServer(function(request, response) {
8      // Parse incoming request
9      const uri = url.parse(request.url).pathname;
10     const filename = path.join(process.cwd(), uri);
11
12     // Check if file exists
13     path.exists(filename, function(exists) {
14         if(exists) {
15             // Read file
16             fs.readFile(filename, function(err, data) {
17                 // Respond with file contents
18                 response.writeHead(200);
19                 response.end(data);
20             });
21         } else {
22             response.writeHead(404);
23             response.end();
24         }
25     });
26 }).listen(8080);
27
28 sys.log('Server running at http://localhost:8080/');
```

Listing 1 *Asynchronous file server created in Node.js. Code modified from source [24]*

The example code 1 shows how a simple file server can be built in Node.js by utilizing

asynchronous callbacks. The example contains three different asynchronous calls: the `http.createServer` call on line 7, `path.exists` call on line 13 and the `fs.readFile` call on line 16. The `http.createServer` callback is called every time the server receives an HTTP request. While waiting for such requests, the process can either move on to other things or idle, if there is nothing else to execute. Once a request is received, we check for the existence of a file with `path.exists`. Since this is an I/O bound task, we will utilize a callback, that will get called once the disk read returns and the Node process is executing I/O callbacks.

If the file exists, we utilize the last asynchronous call of `fs.readFile` to read the contents of the file. Once the I/O operation finishes, we write the received file contents as a response to the initially received request. Node.js utilizes asynchronous interfaces a lot and rarely even offers synchronous counterparts. Blocking execution kills performance in a single process environment.

Node.js uses an event loop to organize the code execution for all the incoming events. The event loop is also used to determine whether the program should continue to run: if there are no active timers or asynchronous I/O calls waiting, the running process may stop. [25]

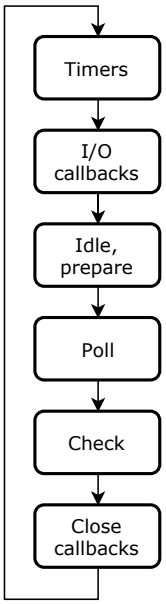


Figure 3.1 *Simplified overview of the Node.js event loop execution order [25]*

Figure 3.1 shows a simplified overview of the Node.js event loop and its execution order. Every phase of the event loop has a First In, First Out (FIFO) queue of relevant callbacks to execute. Generally, in a given phase, the event loop only executes tasks related to that phase and executes callbacks in the queue until the queue is empty or a maximum number of callbacks have been executed. After this, the event loop moves to the next

phase and repeats until there are no active timers or callbacks waiting in the event loop, or the program is otherwise stopped.

Starting from the top, a quick overview of the phases is as follows:

- **timers**: callbacks scheduled by setTimeout and setInterval functions
- **I/O callbacks**: almost every other callback except close, timers or setImmediate
- **idle, prepare**: used internally by Node.js
- **poll**: retrieve new I/O events, execution may block if necessary
- **check**: setImmediate callbacks
- **close**: closing events for sockets etc.

While the asynchronous model utilized by Node.js is powerful and can offer seemingly concurrent execution, it is still executed inside a single process. This means that unlike multithreading, a single Node process may not be able to fully utilize a modern multi-core processor to its fullest. Node.js provides support for clustering and forking of processes, which allows multiple Node processes to share the same socket to distribute work between them. One of the service scaling aspects explored in this thesis later on is dependent on the clustering support of Node.js.

3.2 The Node.js ecosystem

Node.js has amassed a large developer community around it. At the time of writing, the package manager for Node and JavaScript, npm, has around 600000 packages registered [26]. Npm includes both frontend and backend libraries, but it was originally created as a package manager for Node.js.

JavaScript is also a very popular programming language. It is the language with most repositories hosted at GitHub [27], a centralized hosting service for the git Source Control Management (SCM) software. It is also the most popular language for the fifth year in a row and Node.js is the most popular framework in the Stack Overflow 2017 developer survey [28]. Stack Overflow is a popular discussion platform for programming related questions.

While technology choices should not be based on popularity alone, it is a good metric for figuring out how healthy the community is. Even the best languages and technologies need users to thrive, produce libraries and provide support. Having a large amount of existing documentation and discussion to fall back on when facing issues is really valuable.

4. CLOUD BASED HOSTING

Companies have a lot of options for hosting their software. Traditionally, hosting would require purchasing and maintaining own server hardware, which has large upfront costs associated with it as well as a maintenance burden. Purchasing and setting up said hardware is also a painstakingly long process. These days it is much more common to offload some of this maintenance and management burden to third party service providers.

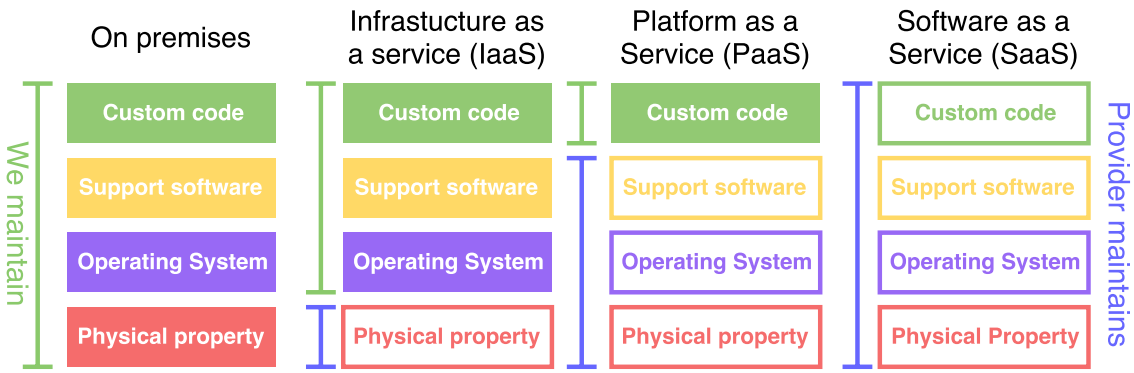


Figure 4.1 *Modern companies have a lot of options when it comes to hosting their software, ranging from self-hosted to fully managed by third parties*

Figure 4.1 shows the breakdown of the four different ways companies can host their software: on premises, Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). The figure also shows how the responsibility is divided between the service provider and the company in the different hosting scenarios.

On premises hosting is the classic way of hosting software. With this option, all the hardware as well as the software is managed by the company itself. Depending on the size of the business, this can be quite a heavy approach. The upfront cost for hardware as well as the continuous maintenance costs are not sensible for smaller companies, though there is a break-even point, where moving from cloud to self-hosted can make sense. Procuring and setting up the hardware is also a time consuming process, that most companies would like to avoid, unless necessary.

Infrastructure as a Service means that the service provider is responsible for the hardware, while the company buying the service has to manage the operating system, supporting

software and the custom code. In this context, supporting software means things like databases or other ready made software utilized by the custom code.

Modern virtualization techniques have made IaaS solutions readily available and quite common. Hosting expenses are quite low, but the developers or maintainers have to pay attention to the whole software stack instead of just their own product. Automating the setup for the hosting environment might require a lot of effort, but it is pretty much necessary when dealing with multiple services instead of a singular monolith.

The next service level up from IaaS has the developers only caring about their custom code. Platform as a Service aims to make the life of developers as easy and straightforward as possible by allowing developers to focus mainly on the application, and not the environment. Due to performance considerations, the hardware layer can not ever be truly ignored, but PaaS providers make common maintenance tasks easier and ease the burden on the development team.

Since the application layer of PaaS providers is often concerned only with the custom code part of applications, PaaS providers need to provide a solution for managing supporting software like caches or databases. These are often provided in the form of additional managed services. PaaS environments provide opinionated options for developers, which can be both a good and a bad thing. Pre-made choices can expedite the development process, but working around platform restrictions can also become painful, if the platform decisions do not fit the project.

The easiest hosting option in terms of maintenance is Software as a Service. SaaS does not require any custom code, instead, everything is handled by the service provider and users can just focus on utilizing the features of the ready made product. This can also be the least flexible choice in terms of customization.

SaaS services often have multiple end users and clearly defined backlogs, so getting new feature ideas or changes through from support to implementation is hard. Often SaaS services are a good choice for things that have existing solutions and that are not a key part of the unique value proposition provided by a service. Being able to offload some of the maintenance burden on other providers makes it easier for the developers.

4.1 Heroku, a modern PaaS solution

Heroku is a Platform as a Service provider, focusing on web applications [29]. Originally founded in 2007, Heroku was later acquired by Salesforce.com in 2010 as a wholly owned subsidiary [30]. Heroku was one of the first cloud platforms available.

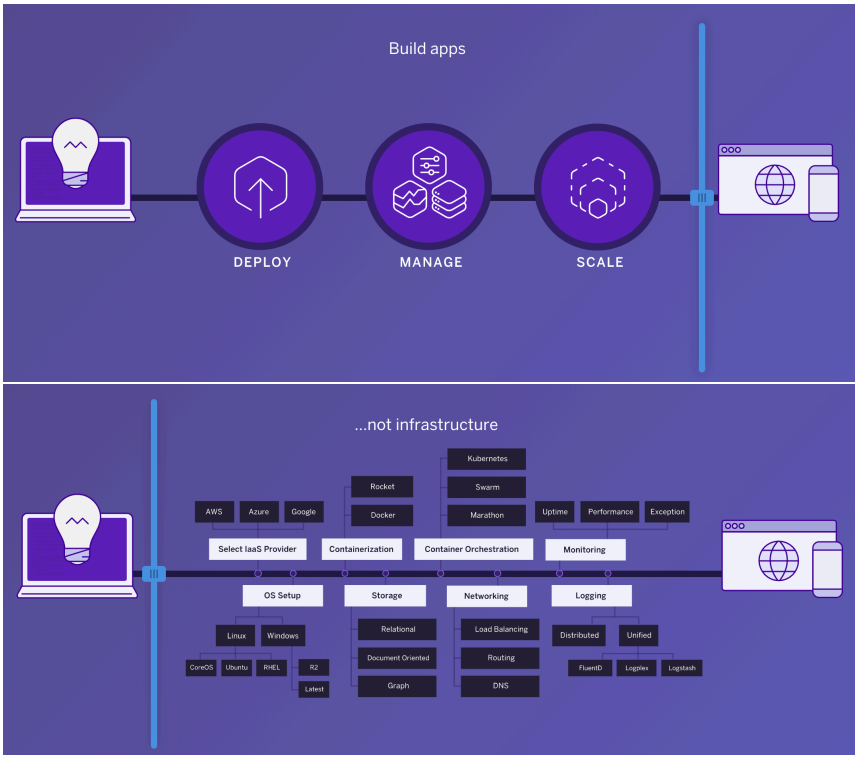


Figure 4.2 *Heroku aims to minimize the time and money spent on Infrastructure and maximize the time spent on the application being built*

As a cloud platform, Heroku wants users to focus on applications instead of infrastructure 4.2. Having proper infrastructure in place is important for application development and maintenance, but it is also an often overlooked area in software development. Infrastructure rarely offers a clear return on investment, which is why its importance might get downplayed. It is important to remember, that just getting the application running is not good enough, it also needs to be maintained. Maintenance benefits immensely from having the proper tooling around it and getting it right from the start is very valuable.

Heroku itself runs on Amazon Web Services (AWS) [31] by utilizing the Amazon Elastic Compute Cloud (Amazon EC2) [32] service on the AWS IaaS platform. Heroku hosts user applications inside lightweight Linux containers, but it also has support for Docker containers.

4.1.1 The Heroku way

As was mentioned earlier about PaaS environments, they can be quite opinionated about the way applications should be built and run. Heroku is no exception in this regard.

One of the co-founders of Heroku, Adam Wiggins, wrote the The Twelve-Factor App [33] guide on how modern web applications should be built. Unsurprisingly, a lot of these

details are also enforced or greatly encouraged by the Heroku platform. The twelve rules of a twelve-factor application are explained in more detail in the following paragraphs.

For Heroku to be able to deploy an application, the **codebase** has to be in a Version Control System (VCS). Not only is this a good practice, Heroku also enforces that applications can only run code that is actually committed to the repository. This way running versions and features in different environments become easier to track. Being able to figure out what is running in development, staging and production should be easy and straight forward, not a mystery to unravel. [34]

Applications should explicitly state their **dependencies**. On Heroku, when an application is deployed, the Heroku platform always builds the container that is going to be ran on the service after the deploy. The environments always start from a blank slate, with code being the only input added at the start. Every dependency needs to be either included with the code or defined through a dependency management system used by the language in question. Examples of such systems include npm [26] for Javascript, Leiningen [35] for Clojure and pip [36] for Python.

The **configuration** for an application is something that is likely to vary based on deployment. Development, staging and production will have different environments and all of these will require their own configuration for things like database addresses, hostnames or third-party service credentials. Storing all of this inside a version control system is problematic for multiple reasons. First of all this exposes private secrets inside the VCS and it also leads to unnecessary commits when managing the environment configuration. On Heroku, configuration is handled through environment variables.

Backing services, like databases or other resources the application needs for it to properly function are referenced with Uniform Resource Locators (URL) or other credentials stored in the configuration. When external services are consumed like this, it allows for easier swapping of service providers, by just changing the corresponding URL. Backing services are called add-ons on Heroku, and we will explore them in more detail later.

A codebase should be transformed into a running application through a three step process: **build, release and run**. When these phases are separated properly, ensuring a consistent and predictable release pipeline becomes easier. During the build stage an application gets transformed or compiled in to an executable bundle that can be ran on the servers. The release stage combines the build with a configuration, making the application ready for the run stage. The run stage is the execution environment for the release stage bundle. [37]

Applications should be broken down in to multiple different **processes**. The processes

should be stateless, and the only data shared between them should be stored in a backing service. The filesystem and memory of a runtime can be used to momentarily store something, but properly built twelve factor applications should not rely on them being available during future requests: when multiple processes are serving the requests, chances are high that a future request will be handled by another process. Heroku cycles dynos every 24 hours, which forces application developers to store everything important outside of the filesystem or Random Access Memory (RAM) of the runtime.

Twelve factor applications listen to incoming requests by **binding to a port**. Applications should not rely on a web server to handle the traffic for them, this is something that will be handled completely inside the application code. For HTTP there are many popular libraries like Jetty [38] for Java or Thin [39] for Ruby to achieve this. Heroku expects applications to be listening on a port defined in an environment variable called PORT. This port binding approach means that applications can also become backing services for other applications through configuration of the consuming service.

Inside a twelve factor application, processes are first class citizens and **concurrency** is achieved by scaling these processes. Services can consist of multiple different process types, for example we might have web processes for handling traffic and worker processes to handle background jobs like messages from a queue. These processes can be scaled individually on Heroku based on the needs of the application. [40]

Application processes are ephemeral and should be considered **disposable**. They can be started and stopped at a moments notice, based on scaling needs or when deploying new configuration changes or versions of the application. This also requires the application to boot up fast so that it can start receiving incoming requests after a few seconds. Processes should also close down gracefully, so that clients do not see interrupted requests from closed processes. This means stopping listening to incoming connections and letting existing connections finish before exiting. This also requires requests made to the application to be short, which is enforced by Heroku by automatically timing out requests after 30 seconds.

Developers often have configurations and environments during development that are completely different from the production runtime. This can lead to unexpected issues that only manifest themselves after going to production. This is why **dev/prod parity** is important in a twelve factor application. If a PostgreSQL database is being used in production, it should also be used in development and staging, instead of something easy and lightweight to setup like SQLite.

Logs are event streams that let us peer into the running state of the application. Traditionally they are stored inside log files, but this is just one output format for logs. Twelve

factor applications should not consider the storage or routing of their log events, the event stream is just written to standard output or stdout. In staging and production environments, it is the job of the execution environment to capture and route the event streams produced by the application. Heroku leaves this job partially to backing services: the logs will be collected and routed by Heroku, but long term storage requires an additional service, since Heroku only stores the last 1500 lines of events produced.

Eventually, every production application needs to have some administrative work done on it, be it database migrations or one time scripts for fixing or cleaning up data. These sorts of tasks should be handled as one-off **administrative processes**. These administrative processes are a part of the application just like the long-running processes, only difference being the fact that they are not on at all times, instead being invoked on demand and running only for a set amount of time. On Heroku this can be accomplished by running one-off dynos, which are spawned with the same application runtime that is used by all the other process types.

4.2 The Heroku environment

Heroku consists of multiple different components, some of which are user controllable and some that are completely handled by the platform. We will go through some of the more important details of the platform, starting with the application containers, which are called dynos in Heroku nomenclature.

4.2.1 Dynos

All applications on Heroku run in a collection of lightweight Linux containers called dynos. Dynos can be put in to three different categories: web-, worker- and one-off dynos. Web dynos are the only dyno types that can receive HTTP traffic, while worker dynos are often used for background processing tasks like message queue workers and timed jobs. One-off dynos are mainly used for administrative tasks. Backing services are never ran inside dynos, dynos are reserved for application code only. [41]

Besides their own container format, Heroku also supports Docker [42]. However, some of the platform features are not either available or do not work as well when running on Docker containers, so the Heroku custom container format is the preferred method of running applications on Heroku. [43]

There are multiple different dyno types available on Heroku, as can be seen on table 4.1. These dynos are differentiated between one another based on memory and computational

Table 4.1 *Dyno types available on Heroku*

Type	Memory	CPU Share	Dedicated	Compute
free	512MB	1x	No	1x-4x
hobby	512MB	1x	No	1x-4x
standard-1x	512MB	1x	No	1x-4x
standard-2x	1024MB	2x	No	4x-8x
performance-m	2.5GB	100%	Yes	11x
performance-l	14GB	100%	Yes	46x

power available, but there are some other, more subtle differences as well. The free and hobby dynos differ from other dynos mainly in the fact that they don not have all of the Heroku runtime features available for them, like application metrics. The free dyno types can also be put to sleep, if the account has ran out of free dyno hours or the application is not actively receiving any traffic.

By default, dyno types ranging from free to standard-2x are ran inside shared instances, meaning that applications from multiple different users are being executed on the same machine. This is why the compute metric is indicated as a range. Also, the share of CPU time or priority the app receives is limited. Performance dynos are ran on dedicated instances, so they always have all of the CPU resources for themselves.

Another option for running smaller dyno types on their own dedicated hardware is to utilize private spaces. Private spaces are a feature intended for larger and enterprise customers, which allows higher level of isolation and security through features like private networking. Inside the common runtime, where applications are being executed by default, every application and backing service is publicly available. Applications in the common runtime can only choose between two zones, the United States or Europe, while private spaces offer more fine-grained control, based loosely on AWS datacenter availability.

Dynos can either be scaled horizontally or vertically. Horizontal scaling happens by adding more dynos of the same type. By default, this is always initiated by the developer or the maintainer, but there are some add-ons that can offer autoscaling features based on metrics like traffic, time of the day or load. Heroku also offers autoscaling features for their performance dyno types. Vertical scaling is achieved by moving to larger dyno types.

Dynos are controlled by a dyno manager. All non-enterprise customers use a shared dyno manager for each availability zone. An availability zone is a region that Heroku provides server capacity for. The shared dyno manager means that Europe and US regions have different dyno managers. Private spaces, which is feature for enterprise customers, has a

dedicated dyno manager. The dyno manager is responsible for running automatic health checks on the application, deploying additional dynos when requested by user or autoscaling and doing different kinds of maintenance tasks, trying to make the application run as free from unnecessary interruptions as possible.

Slugs

During deployment, Heroku applications get compiled into slugs, which are then handed off to the dyno manager for execution. Slugs are a compressed and pre-packaged copies of the applications. The slug compilation process begins when a new deployment is initiated by a git push command to the Heroku remote git. The slug compilation process downloads, builds and installs application dependencies based on the defined or detected language and tooling. [44]

Slugs have a maximum size limit of 500 megabytes of disk space. Smaller slugs are preferable, since this makes creation of new dynos faster due to faster transfers of the slugs to the machines that are going to be running the new dyno instances.

4.2.2 Buildpacks

Buildpacks are responsible for transforming an application code repository into a deployable slug. Buildpacks are compromised of a set of scripts, which, depending on the programming language, retrieve dependencies, output assets, compiled output, and more. Officially, Heroku supports Node.js, Python, Ruby, Java, Clojure, Scala, PHP and Go. [45]

By default, the Heroku deploy process tries to automatically match the project with a matching buildpack by checking one by one, until a matching buildpack is found. If a matching buildpack is found, it is permanently marked as the matching buildpack for future builds. Users can also explicitly define the required buildpack through the Heroku Command Line Interface (CLI).

The buildpacks are built upon the platform base images, which Heroku calls stacks. These stacks are maintained by Heroku and they are based on the Ubuntu Linux distribution [46]. While Heroku provides buildpacks for a limited set of languages and frameworks, users can build custom buildpacks to support virtually any language and framework that can run on Linux. The official buildpacks are provided as open-source for reference on how new buildpacks can be created.

4.2.3 HTTP Routing

In Heroku, HTTP traffic is automatically routed to the hostnames associated with the web dynos of an application. On the common runtime, every application gets a herokuapp.com sub-domain associated to them. All web dynos that are not located in a private space are publicly available. [47]

Inbound HTTP requests are received by load balancers that can do SSL termination for Hypertext Transfer Protocol Secure (HTTPS) traffic. The load balancers hand off the requests to Heroku routers, which determine the location of the corresponding web dynos and forward the traffic. Routers distribute requests randomly between available web dynos.

For normal HTTP requests, routers will wait up to 30 seconds to receive an answer from the web dyno the request was forwarded to, after the connection was accepted. If an answer is not returned within 30 seconds, the router times out the connection and replies with an error to the client. Servers can extend this timeout window by utilizing long-polling, where every byte returned from the server resets a rolling 55 second window in which the server needs to send more data or end the request before the router times out.

4.2.4 Add-ons

Since dynos are only meant for application code, Heroku provides backing services like databases through add-ons. These add-ons get attached to specific Heroku applications and they most often expose themselves or their configuration through environment variables. [48]

The add-ons available range from databases to application management services to analytics to testing: a lot of different services are readily available, for those that are willing to pay the price. Some of these add-on services are managed by Heroku themselves, but most of them are handled by third-party service providers.

4.3 Conclusions

Heroku is a PaaS, which offers a lot of expensive to build infrastructure built in. It also aims to make application management as simple as possible. All of this ease of use and ready built infrastructure does come with a cost, however.

Since every process runs in its own dyno, a simple app can easily rack up a few dynos. Each dyno is paid for by the hour based on how long the dyno was running in a given

month, and even a few dynos start to add up, especially on larger dyno types. Add-ons are also paid for, with only the most simple plans being offered for free. All in all, hosting services on Heroku is more expensive than setting up a virtual server or multiple virtual servers to run on.

But building infrastructure is also expensive and a time consuming task. The fact that Heroku gives a lot of it upfront and makes it possible to focus mainly on the application itself is really valuable in projects, where visible results are needed fast. On shorter projects, the time spent on building comparable or worse infrastructure than what Heroku offers out of the box may never pay itself off during the lifetime of the application.

The choice between a PaaS service and self built infrastructure comes down to time and cost. In the long run, self built infrastructure will be cheaper, but that approach will also incur heavier costs upfront. Costs, which depending on the lifetime of the project being worked on, might never be paid off by the savings in runtime costs.

5. PRODUCTION SERVICE ISSUE

We have gone through what microservices are, what Node.js is and what the ecosystem is like, and what the Heroku PaaS offering looks like. These are all relevant factors in the case of our production service, since it is programmed in Node.js, hosted on Heroku and it is split into multiple independent services.

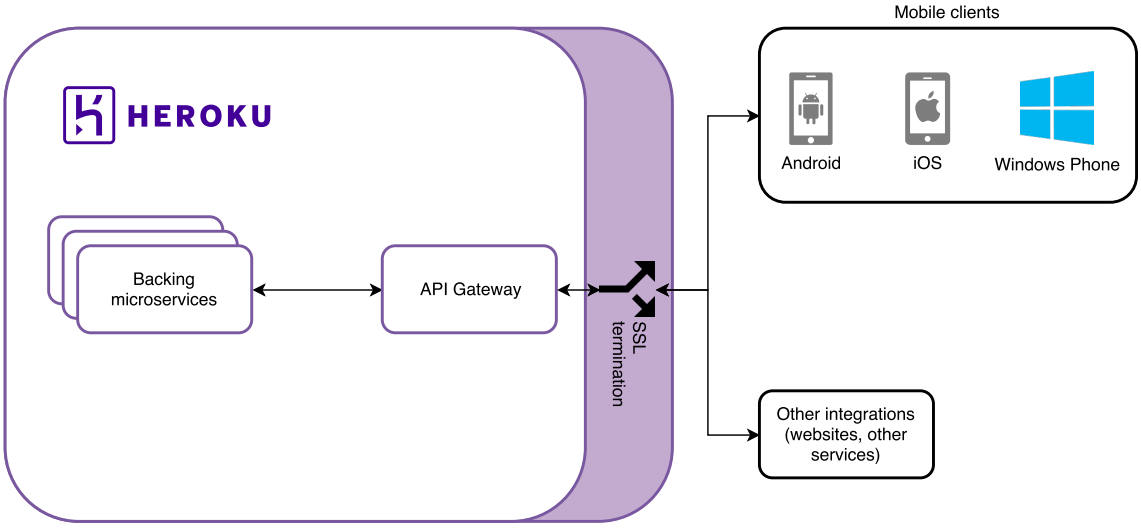


Figure 5.1 *Simplified view of the overall architecture for our production application. The mobile platforms are the main consumers for the service, but we also have additional consumers in the form of websites and other services*

Figure 5.1 shows a simplified view of the overall architecture for the service. Our main API consumers are the mobile applications. To simplify the usage of our microservices, we have also created an API gateway to coordinate the API calls between different services and to route traffic.

In addition to the mobile applications, we also have some third party websites and services utilizing our APIs. Some of them use the microservices themselves directly, while others go through the same API gateway the mobile applications use. Because the microservices are hosted on the common runtime of Heroku, they are all publicly available, although they do require authentication credentials for actual use. Heroku supports private networks, called private spaces on Heroku, only on enterprise customer levels.

Besides the main API gateway created for the mobile applications, we also have some additional, smaller gateway services. These offer limited access to a subset of our microservices for some of our integrations. This allows us to manage the incoming traffic better, while also enforcing finer-grained access to our API consumers.

5.1 Push notifications

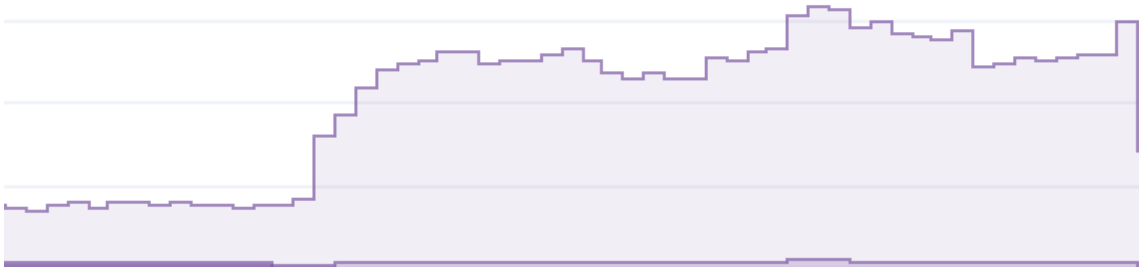
On mobile platforms, push notifications are a popular way to increase user engagement. Push notifications are messages that are sent from a central server to end users' devices. These notifications are application specific and can have different customizations depending on the mobile platform. Typically the message being shown to the user and the application view associated with that message can be defined when sending a notification, but even more customization options might be available.

Every major mobile operating system has its own service for sending and receiving push notifications. iOS has Apple Push Notification service (APNs) [49], Android uses Google Cloud Messaging [50] and Windows Phone has Windows Push Notification Services (WNS) [51]. Implementing logic for interfacing with all of these services separately in a multi-platform application like ours would require a lot of work, but luckily there are services that take care of this complexity for us.

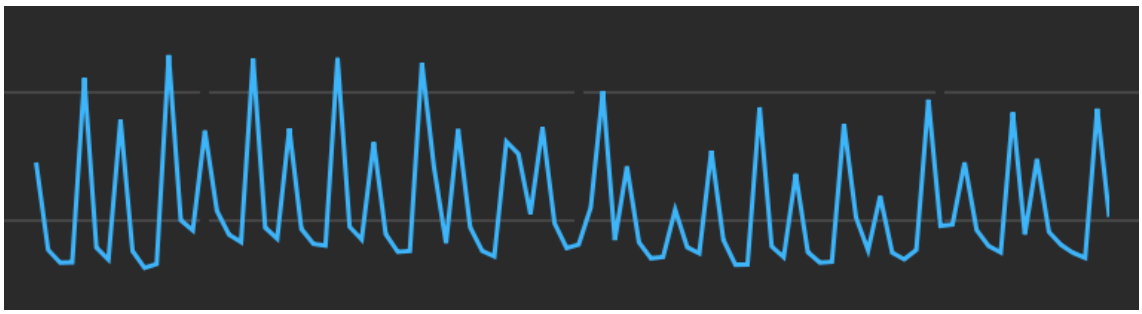
5.1.1 Notifications causing peaks in traffic

Push notifications are often sent either as a reaction to something that an end user did or as general announcement messages. In our applications, users can share shopping lists between one another. The recipients of the share get a push notification that notifies them about the share that occurred, and that opens the shared list when clicked. Since these lists are often shared between very small groups, the traffic caused by user initiated notifications are minimal. However, general announcement type notifications to all users or user segments consistently cause high loads on our servers.

When a notification is sent from the servers to the clients, the user sees a notification prompt with the application logo and the message that was sent. The sent notification does not send requests to our servers, but once the user clicks the notification and loads the application, our servers start to receive traffic When many users do it at the same time, we can see clear peaks in traffic to our services. Of course, the meaning of the notifications is not to generate traffic, but that is a clear side-effect that happens every time a batch of push notifications is being sent.



(a) Gateway traffic pattern when push notifications start



(b) Daily visitors exhibiting clear spikes on days with push notifications

Figure 5.2 *Gateway load and daily visitor graphs that demonstrate the effect of push notifications*

Figure 5.2 shows metrics from our gateway service, demonstrating the effect that push notifications can have. Almost immediately after a push notification batch has been initiated, we see double to triple the normal amount of traffic, as seen in figure 5.2a. This trend continues until the sending of push notifications eventually stops. The effect of push notifications can also be seen in figure 5.2b, which shows the amount of daily visitors.

When this service was initially launched, we sent all push notifications at once. SNS can handle a lot of concurrent notifications being sent, but that does not necessarily mean that our service could handle all of the incoming traffic. This effectively caused us to do a Denial of Service (DoS) attack on our own service. While every user does not click the notification as soon as they receive it, the clear majority of users interact with their notifications quite fast. Since we were sending the notifications as fast as SNS would accept them, we ended up generating way more traffic than our server capacity could handle.

Our initial fix to the traffic peaks was to do the notification sending in batches. Instead of sending all of the notifications as fast as we can, we send them a batch at a time while waiting a moment between every batch. While this does mean that it takes longer to finish sending the notifications and there is a large difference in the time different users get the same notification, this allows us to manage the traffic a little bit better.

5.2 Consistent high load is problematic in a shared environment

Our fix for the push notifications caused traffic to have smaller peaks, but it also made the traffic stay higher for a longer period of time. This in turn means that our servers need to be able to sustain double to triple the normal amount of traffic over a long period, until the push notification traffic eventually dies down.

Originally we used the standard 1X and 2X dyno types from Heroku to host the gateway. While our initial fix for the push notifications helped at the start, once the amount of users grew, we started to see a new issue in our gateway service. During consistently high loads, like the ones we would see during push notifications, our gateway would start timing out connections.

One solution to ease the load on the gateway would have been to adjust the size of the notification batches or the frequency of them. With our existing batch size and sending frequency, sending a single notification to all of our users took around seven to eight hours. We did not want to slow down the notification sending process any more than necessary.

The standard dyno types are hosted in the common runtime by default. This means that they are hosted on machines that are sharing resources with other users of the Heroku common runtime. While Heroku tries to make resources available as evenly as possible, there is nothing stopping a single service from hogging up a significant portion of the resources available. It is unlikely that this service can keep on going for longer periods, but momentarily it may cause issues for other customers. As we can see from the table 4.1, the performance levels for standard dynos is indicated as a range. This is both due to the performance depending on the implementation, but also being affected by available resources at the time of execution.

While our application does not have strict Service Level Agreements (SLA), dropping requests randomly is non-ideal and something we would like to avoid. Even if this did not affect all of our users, we wanted to explore potential solutions for the problem. Eventually, we tracked down the issue to insufficient resources at peak traffic coupled with the potential for noisy neighbors of the shared environment to cause even more issues when we are already at max capacity. While we could have easily mitigated the traffic by adding more standard dynos, that would not help with the shared environment issues. We also needed to figure out what sort of server configuration would be reasonable for the amount of traffic we were seeing.

One option to dealing with noisy neighbors is to optimize the implementation, so that the CPU usage stays below maximum. Depending on the application, this can be quite time consuming, and it might also turn out that the application is already quite well tuned and

additional hardware is needed to match the demand. Another option is to move away from a shared environment, so that we get better guarantees for our level of performance. Heroku offers the premium dyno types and the private spaces for this purpose.

Just changing the dyno type to premium would not really work in our situation. Due to the single threaded nature of the Node.js runtime, we would not have been able to utilize all of the available computing resources of the larger dyno types. Node.js offers clustering to solve issues in vertical scaling.

6. CLUSTERING NODE.JS TO IMPROVE VERTICAL SCALABILITY

In chapter 3, we mentioned the single threaded nature of Node.js. On smaller servers or dyno instances in Heroku, a single Node.js process is enough to utilize all of the available resources. The available resources on larger instance types will be wasted, unless Node.js clustering is implemented.

Clustering [52] allows Node.js processes to spawn and manage child processes that can share incoming work between them through a shared server port. While clustering has been made mainly with networking in mind, it can also be used for other use cases where multiple worker processes are needed.

```
1  const cluster = require('cluster');
2  const http = require('http');
3  const numCPUs = require('os').cpus().length;
4
5  if (cluster.isMaster) {
6      console.log(`Master ${process.pid} is running`);
7
8      // Fork workers.
9      for (let i = 0; i < numCPUs; i++) {
10         cluster.fork();
11     }
12
13     cluster.on('exit', (worker, code, signal) => {
14         console.log(`worker ${worker.process.pid} died`);
15     });
16 } else {
17     // Workers can share any TCP connection
18     // In this case it is an HTTP server
19     http.createServer((req, res) => {
20         res.writeHead(200);
21         res.end('hello world\n');
22     }).listen(8000);
23
24     console.log(`Worker ${process.pid} started`);
25 }
```

Listing 2 *Example of Node.js clustering of a web server. Code copied from source [52]*

The code example 2 shows how a simple web service can be clustered. When we are clustering an application, we have two types of processes: the master process and the forked child processes. This is represented in the example code by the if-statement beginning on line 5. The master process is responsible for creating the forked child processes as well

as managing their lifecycle. The child processes go to the else branch of the if-statement, and launch a web server. In this example, we create as many worker nodes as there are CPU cores on the machine executing the code.

The workers are spawned by using the `child_process.fork()` method. The fork method is special, in that it is used to spawn new Node.js processes [53]. The spawned process has a built-in channel for communication between the new process and the one that spawned it. Besides this Inter-process communication (IPC) channel, the child processes are completely independent from the one that spawned them. This means that every process has its own memory and their own V8 instance. The created IPC channel is how the processes can share the listening sockets that they create with the master process.

By default, the master process in clustering is responsible for listening on the port and dividing the work between the child processes in a round-robin fashion. The master process also avoids overloading any single worker. There is a secondary operation mode available, where the master process creates the socket and sends it to the child processes. The default method is the recommended one, since the alternative approach can have very uneven performance characteristics depending on the machine the program is running on.

As we can see in our code example 2 on line 13, the master process can listen and react to some child process events, most notably process exits. By default, the clustering module does not manage the worker pool in any way. If all workers have exited, the master process just stops accepting incoming connections, but it does not restart crashed processes by default. This is something that has to be managed by the end user.

6.1 Implementing clustering in an existing web application

Since we are implementing clustering for a web application, the implementation details themselves are quite simple. We used the default clustering approach, where the master process distributes work between the child processes. We did not use the raw clustering API, but instead opted for an NPM library building on top of it called throng [54].

Code example 3 shows how we use throng in our API gateway service. The main reason we chose throng for our clustering implementation, was that it takes care of the child process lifecycle for us. We could even define process lifetimes if we wanted to, but in our use case all of the workers should be up and running until the process is restarted or stopped. Throng restarts crashed processes automatically. Ideally, crashes should not happen, but they are nicely taken care of by the library itself.

Starting from line 6, we can see how throng is configured. We define the amount of work-

```
1  const createApp = require('./index');
2  const config = require('./config');
3  const http = require('http');
4  const throng = require('throng');
5
6  throng({
7      workers: config.WEB_CONCURRENCY,
8      lifetime: Infinity,
9      start: startServer,
10     grace: 10000, // Grace period for shutting down workers
11 });
12
13 function startServer(id) {
14     const app = createApp();
15     const server = http.createServer(app);
16
17     server.listen(config.PORT);
18     server.on('listening', function serverListening() {
19         logger.info('Server worker %d listening on port %d', id, this.address().port);
20     });
21 }
```

Listing 3 *Clustering with the throng library*

ers we want to spawn, the lifetime of the workers, the function that the child processes should run, as well as a grace period for the processes when we are shutting down the service. The `config.WEB_CONCURRENCY` configuration value is something that is provided by Heroku, and it is based on the available memory on the dyno type that is being used [55]. We can define the `WEB_MEMORY` environment variable to configure how much memory we want to allocate per concurrent process. Heroku then defines the concurrency value based on the total memory available and our planned memory allocation.

Inside the worker start function `startServer`, we first initialize the Express [56] web application by calling `createApp` on line 14. This initializes the routes that are served by the gateway, authentication and other business logic required for the different API endpoints. The Express constructor returns a function that we can pass on to Node.js' `http.createServer` function. Once we call the listen method for the server object that gets returned, our processes are ready to receive connections. The `PORT` configuration value is provided by Heroku, and it is the only port through which Heroku will forward traffic through to the web application. This value can change on every restart, which is why the application reads it from the environment variables.

Implementing clustering in an existing application required minimal changes, since it can be fully implemented at the entry point of your application. We only modified the way our application is initialized and added some additional configuration values to it. After these changes, we could deploy the clustered version of your API gateway and manage the number of running workers by configuring the `WEB_MEMORY` environment variable. We went with 512 megabytes of memory per process as the baseline, which allows us to run five concurrent processes on a premium-1 dyno.

7. EVALUATION OF THE PERFORMANCE IMPACT

Performance testing of an API gateway service is complicated. Since the service is mainly responsible for doing request proxying to the underlying microservices, we would have to generate a lot of traffic to start to see the benefits of clustering. We created a simple example service [57], so that we could explore the effects of clustering on response times in a more controlled manner.

The example service is a Node.js based application written in TypeScript [58], which is a superset of JavaScript that adds typing support. It is a simple application that has only two endpoints, both of which do a simple data fetch from a PostgreSQL [59] database and return the result as JSON. The implementation for the controller layer, responsible for request input validation and HTTP responses, can be seen in appendix A. The clustering implementation is shown in appendix B.

7.1 Performance testing with Gatling

We used Gatling [60] to create the performance tests used to measure the effects of clustering in our example application. Gatling offers a Scala based domain-specific language (DSL) for test implementation. We used two different tests for our example backend: one for each API endpoint. The test contents are shown in appendices C and D. Only thing different about the tests is the API endpoint being called. Every simulated user fetches a random data item from the backend.

Table 7.1 *Test system specifications*

Laptop model	15” Macbook Pro, Mid 2015 model
CPU	Intel Core i7 2,5 GHz
Memory	16GB 1600 MHz DDR3
Node.js version	8.11.1

The performance tests were ran locally, with the test system running both the load gener-

ating tests with Gatling and the backend service under test. Key specifications of the test machine are described in table 7.1. Since we are sending and receiving traffic on the same host, these tests trivialize network latency. In a more realistic environment, the database server, application server and load generating application would most likely be running on different machines. While our test setup is not ideal, it is enough to demonstrate the effects of clustering on more CPU intensive workloads. We ran the Gatling tests for non-clustered and clustered versions of the application, and compared the results. Since we were using a four core CPU, we clustered the application to four processes.

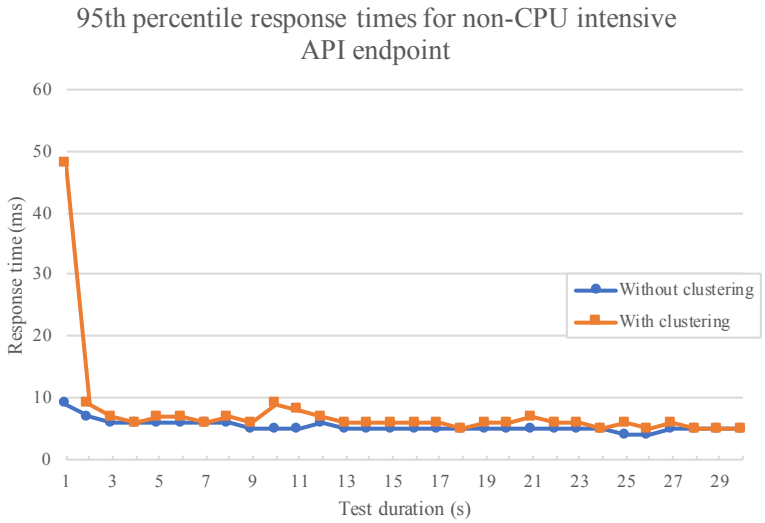


Figure 7.1 *95th percentile application response times for the non-CPU bound API. The clustered version of the application performs a little worse than the non-clustered one*

Figures 7.1 show the response times for the non-clustered and clustered versions of the non-CPU intensive API. In the non-clustered version response times, we can see some peaks, but overall it is very similar to the clustered results.

Table 7.2 *Non-CPU bound API test response times in more detail*

	Non-clustered	Clustered
Minimum (ms)	2	2
50th percentile (ms)	4	4
75th percentile (ms)	5	5
95th percentile (ms)	5	7
99th percentile (ms)	7	8
Maximum (ms)	20	48
Average (ms)	4	5
Standard deviation (ms)	1	3

When we look at the response times shown in the table 7.2 in more detail, we can see that the difference between the two test runs are minimal. In fact, the clustered version seems to be ever so slightly slower that the non-clustered one.

The reason we are not seeing any benefits from clustering with this workload is the fact that it is not CPU bound. Since we are sending and receiving traffic on the local network and doing very minimal data transformation for the responses, we can easily serve all of the traffic that the tests generate. To get anywhere near 100% single core CPU usage would require a lot more traffic. The reason the clustered application is performing worse than the non-clustered one is because of the additional process doing the round-robin work distribution. With the test workload, the added latency is minimal, but it is still there.

To emulate a more CPU intensive workload, we added another route to our example back-end that is almost identical with the one we already tested, but it also has a busy loop in it. The busy loop is just an empty for loop, where the CPU will spend lots of iterations on every request. While not a realistic representation of actual CPU intensive work, a busy loop is enough for our testing.

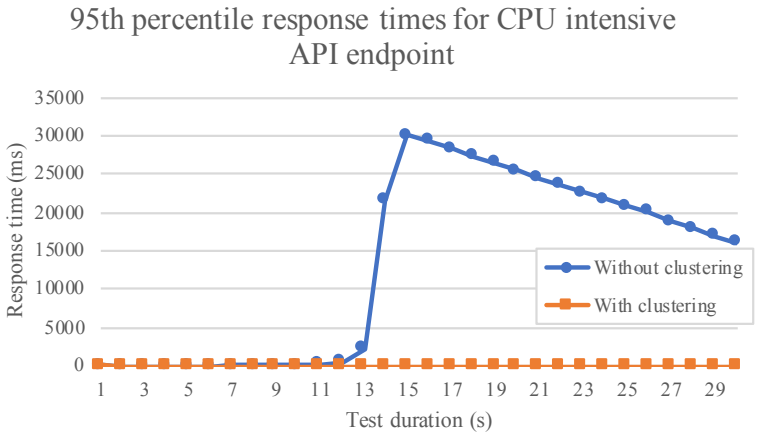


Figure 7.2 *95th percentile application response times for the CPU-bound API. The non-clustered version of the application starts to slow down once there are enough simultaneous requests*

Figure 7.2 shows the response times for the CPU bound API tests. Here we can see the non-clustered application response times keeping up with the load at first, but the response times start climbing once a certain threshold of concurrent users is reached. The response times stay high until there are no new users being added, and the application has the chance to clear the backlog of requests. On the clustered application side we see smooth response times for the whole duration of the test. Since we are spreading the load to multiple cores, we do not get the same issue of queued up requests waiting for their responses.

Table 7.3 *CPU bound API test response times in more detail*

	Non-clustered	Clustered
Minimum (ms)	32	31
50th percentile (ms)	18834	34
75th percentile (ms)	23793	35
95th percentile (ms)	28688	37
99th percentile (ms)	29856	41
Maximum (ms)	30144	80
Average (ms)	16392	34
Standard deviation (ms)	9842	3

Table 7.3 shows the CPU bound workload response times in more detail, and we can see the non-clustered application was clearly struggling to serve the traffic it received. Most users would have most likely stopped waiting for a response after a few seconds, and on Heroku, our connections would have been forcefully ended by the router after thirty seconds of waiting.

While the penalty of clustering is minimal and the implementation details are really simple, we can clearly see that more CPU bound loads benefit a lot better from spreading the load across multiple cores. While request proxying on the API gateway is not CPU intensive, everything starts to add up once you have enough traffic.

7.2 Clustered gateway in production

Since the actual implementation details of the clustering were quite simple, we were able to do the necessary changes fairly quickly. This allowed us to effectively use the larger instance types provided by Heroku. By default, Heroku recommends using clustering on standard-2x type dynos and up. [61]

After we deployed the application, we scaled it up to performance-1x dynos. This allowed us to cluster up to five Node.js processes per server. Because the gateway application is mostly proxying traffic, the memory footprint of each individual Node.js process is minimal, since memory is mainly allocated for requests being handled by the server. Unfortunately, the CPU resources are in much higher demand, and once we reach that maximum of allocated CPU resources, we start to see timed out connections.

Since the API gateway needs to be able to handle the combined traffic of all of the services behind it, it has to perform well. Because gateways mostly just proxy traffic, even simplistic implementations are usually good enough, but once the servers need to be able

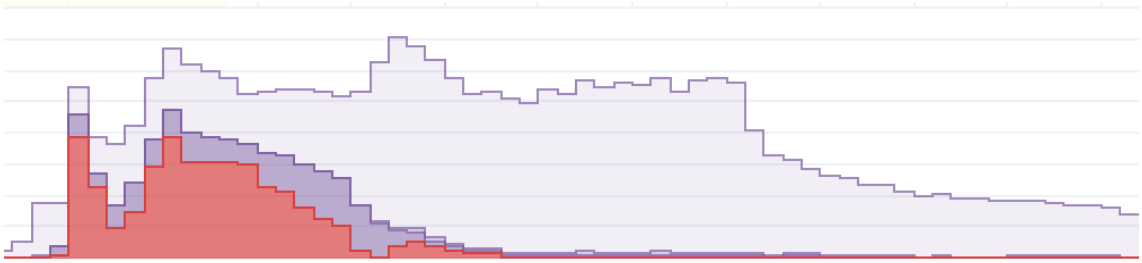


Figure 7.3 *Peak traffic on the API gateway. This traffic was about five times larger than our usual peak traffic, which quickly lead to timed out connections due to low capacity. Red portions of the graph describe HTTP internal server error responses. The darker purple is used to denote HTTP client error responses*

to handle hundreds of requests in a second, even small delays in processing can become huge bottlenecks. This can be seen in figure 7.3, which shows a snapshot of extremely high traffic occurring on our API gateway. During a timespan of around 30 minutes, our notification servers erroneously sent as many notifications as fast as they could, which lead to our API gateway receiving around five to six times higher peak traffic than we would have normally anticipated.

This overflow of traffic leads to our gateway timing out connections. After the first ten minutes of the peak, we manually scaled the servers up to be able to meet the demand. After we scaled the API gateway up, the timeouts stopped. Our backing services were able to meet the demand, the main issue was just our API gateway being scaled too low for the amount of traffic. While Heroku offers autoscaling for performance dynos [62], which we were using at the time of the traffic spike, it does have the downside of taking a while to react to large traffic spikes. When it eventually did start to scale up, we had set the maximum scaling limit too low to be able to handle all of the incoming traffic.

Clustering gave us the ability to move up to larger instance types and take autoscaling in to use. While useful, no autoscaling solution is perfect. The one provided by Heroku is a bit problematic in our gateway use case, since it scales based on the 95th percentile response times. Because the gateway is mainly a proxy, response times can vary based on backing services and their integrations, so slower response times are not necessarily due to degrading performance or not being able to meet the demand. Still, having the autoscaling is a helpful, since we are only actively monitoring the service during work hours.

With increased traffic either through peaks or just organic user growth, we still need to manually scale up our services to be able to meet the normal level of demand. There is no automated system that will do this perfectly. Adding more capacity is trivial in Heroku, so for our use case this mostly comes down to investigating traffic patterns and reacting accordingly.

8. CONCLUSIONS

At the start of this thesis, we introduced the following research questions that we wanted to find the answers for:

1. What are microservices and how to build and manage them?
2. What limitations does our environment impose on our services and scalability?
3. How did we fix our API gateway scalability issue?

We will go through each of these questions in more detail, evaluating the answers our research found for the questions. Finally, we will represent possible future optimizations and changes to our architecture, that can make scaling for peak loads easier or possible with less hardware.

8.1 Microservices

Before we started the research for this thesis, we already had an existing service utilizing microservice architecture and an API gateway pattern running in production. To fully understand our scaling problem on the API gateway level of the service, we first wanted to know more thoroughly what microservices are and how they can be built and managed. This research is represented in chapter 2.

Based on a literary review and research, we found out that the definition for microservices is a fairly loose one, mainly due to the fact that there are many ways to implement the architectural pattern. In essence, it is an architectural pattern in which a service is built as a collection of independent, small services, each responsible for a single piece of functionality. Services done in the microservice architecture pattern are distributed services which communicate over the network with different synchronous or asynchronous communication mechanisms, like REST APIs or message queues.

The API gateway pattern we used is a common method of exposing the functionality of a microservice based architecture. The main alternative for this is service discovery, in

which a centralized system keeps track of the different services and their addresses. Other microservices and clients can then query the service locations from that centralized location, after which they can call the service directly.

The service discovery method would have been one effective method against our scalability issues, but it would have also introduced additional complexity on the client side. In the API gateway pattern, clients effectively call a single API, which proxies requests to the individual services. Service discovery forces clients to know some details of the underlying complexity of the system being used. Since the centralized service for discovery is mainly a key-value store, performance is unlikely to be an issue even with a large amount of calls being done every second.

Overall, we are satisfied with the architecture of our service, excluding the issues at peak loads. If we were able to redo the whole architecture from the ground up, we would most likely be more strict with the API gateway as a proxy, and limit the amount of custom functionality it has. To ease up the peak loads for our main gateway, we have already begun creating smaller, more specialized gateway implementations, that allow access to a subset of services. These gateways often use different authentication methods than our main one, which is also one reason why they are new and independent services.

8.2 Environment limitations

Both our execution environment and hosting provider impose some restrictions on services, which we had to explain in more detail to make the scalability issue clearer. Our execution environment, Node.js, is discussed in more detail in chapter 3. Our hosting and hardware provider Heroku is explored in chapter 4.

Node.js is a JavaScript based runtime. Typically JavaScript has been a programming language used in browsers, but Node.js allows server side execution of JavaScript code. The main reason we used Node.js is that in our company a lot of people know Node.js, so it is easier to rotate people in and out of the project. Technically microservices allow for every individual service to be programmed in a different language, but there are benefits to choosing a main language that the services utilize. For example, if every service exposes a REST API, it is likely that some common patterns will surface that can then be shared as libraries between services.

The main limitation that Node.js has is the single-threaded nature of its execution. All user code is ran inside a single thread by default. This means that in a multi-core environment, a single Node.js process is unlikely to utilize all of the available computation capacity.

Node.js offers the clustering module to fix this issue. A single Node.js program can be clustered to be ran inside multiple processes, which allows for more effective use of available resources in a single machine. Clustering is fairly easy to implement and solves the vertical scalability issues that Node.js has.

Heroku enforces a set of rules on every application running on its platform. For example, a request can last a maximum of 30 seconds, after which it is terminated by the platform. This forces developers to think more thoroughly about synchronous and asynchronous calls to services. Most of the machine types offered are also ran in a shared environment, which means that a single machine is running multiple different services from different customers. This can introduce the issue of noisy neighbors, where other services running on the same machine are hogging most of the resources. Services can also run out of their allocated CPU time, meaning that performance can degrade for a while if there are larger peaks in CPU utilization.

The premium dyno types offered by Heroku run on dedicated hardware. This was one of our motivations to get rid of the vertical scalability issues, since dedicated hardware would rule out issues with noisy neighbors. Noisy neighbors are most likely a minor issue, but at peak loads everything starts to add up quickly.

8.3 Fixing API gateway scalability

The issue with our API gateway timing out connections is caused during peak loads. These peak loads are in turn caused by push notifications being sent to all of our clients. Normal service traffic is something that we can handle easily, but peak traffic really pushes our API gateway to its limits. The issue is explored in more detail in chapter 5. The actual implementation details of the clustering are shown in chapter 6.

We wanted to scale up to the dedicated instance types provided by Heroku, which meant that we had to implement clustering on our API gateway service. The clustering implementation itself was trivial and was deployed to production quickly.

Scaling vertically to larger instance types did not remove the issue of timeouts completely. Moving to the premium instance types of Heroku clearly improved the amount of requests we could process, but we can still receive too many requests, which cause timeouts to happen. The premium instance types also allowed us to utilize the autoscaling feature provided by Heroku. While helpful, it often takes a while to react to increased load and is not fast enough for really sharp increases in traffic. It uses response times as its metric for scaling, which causes a lot of unnecessary scaling events on our API gateway service.

There is also the issue of optimizing costs with these new dyno types. Previously the API gateway used standard-2x dynos, costing 50 dollars a month per dyno. Performance-m dynos, which the service is currently utilizing, cost 250 dollars a month per dyno. A single performance-m dyno can be clustered up to 5 processes per dyno, if every process is allocated 512 megabytes of memory. A standard-2x dyno can be clustered to two processes per dyno, if we are leaving every process with 512 megabytes of memory. There is a clear premium here being paid for the dedicated hardware, since running an equivalent amount of standard-2x dyno processes would cost 150 dollars a month.

We think that the dedicated hardware makes it easier for us to gauge the performance of the API gateway. Additionally, the autoscaling is helpful, especially since the service is not monitored outside of work hours.

8.4 Future work

The clustering implementation allowed us to do vertical scaling, but it did not fully solve the core issue. While we can process more traffic, we will still eventually hit a cap after which our API gateway starts to time out connections. It is only natural that higher loads eventually require more hardware to deal with, but we feel like there are further optimizations that could be done on the gateway level to improve performance further and possibly save on costs.

Currently the API gateway is configured to parse all of the JSON payloads of every request. In a normal API service this is a sensible thing to do, since you save a lot of boilerplate code not having to parse the JSON payloads inside every route individually. But in a service that mostly proxies traffic forward and relays the responses, this leads to wasted CPU cycles. Parsing JSON is most likely well optimized in Node.js, but when processing hundreds of requests per second, even smaller computations can start to add up.

The main reason this configuration change has not been done or tried yet, is that there are some routes that actually use the payloads of incoming requests. The change would then require more thorough investigation than just a simple configuration edit.

Additionally, the instance types being used could be optimized further. A performance-m dyno can be clustered to 5 processes, but a performance-l dyno clusters up to 28 processes and costs 500 dollars a month. If one or two performance-l dynos were enough to handle all of the traffic going through our gateway, we could get some cost savings by scaling even further vertically. This would require mainly analyzing existing scaling patterns, their performance and further testing.

Another option would be to completely rewrite the API Gateway in another language. A language like Go or Rust could result in a more efficient gateway implementation. This would require a lot of testing before deployment, and would increase the maintenance burden of the development team due to a different language being used.

BIBLIOGRAPHY

[1] M. Nygard, *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007, ISBN: 0978739213.

[2] M. Kavis, *Architecting The Cloud: Design Decisions for Cloud Computing Service Models*. Wiley, 2014, ISBN: 9781118617618.

[3] R. McMillan and R. King, "Netflix to Move Fully to 'Cloud'", English, *Wall Street Journal*, p. 18, 2015, ISSN: 09219986.

[4] *Microservices - Martin Fowler*, Accessed 2.5.2018. [Online]. Available: `https://martinfowler.com/articles/microservices.html`.

[5] S. Newman, *Building microservices : designing fine-grained systems*. Sebastopol, CA: O'Reilly Media, 2015, ISBN: 978-1-4919-5035-7.

[6] C. R. with Floyd Smith, *Microservices: From Design to Deployment*, Accessed 24.10.2017, 2016. [Online]. Available: `https://www.nginx.com/resources/library/designing-deploying-microservices/`.

[7] *An overview of RMI applications*, Accessed 25.10.2017. [Online]. Available: `https://docs.oracle.com/javase/tutorial/rmi/overview.html`.

[8] *Apache Thrift - Home*, Accessed 25.10.2017. [Online]. Available: `https://thrift.apache.org/`.

[9] *The Eight Fallacies of Distributed Computing*, Accessed 25.10.2017. [Online]. Available: `http://nighthacks.com/jag/res/Fallacies.html`.

[10] *Understanding HATEOAS*, Accessed 2.5.2018. [Online]. Available: `https://spring.io/understanding/HATEOAS`.

[11] *RabbitMQ - Messaging that just works*, Accessed 7.11.2017. [Online]. Available: `https://www.rabbitmq.com/`.

[12] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services", *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, ISSN: 0163-5700. DOI: `10.1145/564585.564601`. [Online]. Available: `http://doi.acm.org/10.1145/564585.564601`.

[13] *Consul by HashiCorp*, Accessed 8.11.2017. [Online]. Available: `https://www.consul.io/`.

[14] *Amazon API Gateway*, Accessed 8.11.2017. [Online]. Available: `https://aws.amazon.com/api-gateway/`.

[15] *Kong - Open-Source API Management and Microservice Management*, Accessed 8.11.2017. [Online]. Available: https://getkong.org/.

[16] S. J. Fowler, *Microservices in Production*. Sebastopol, CA: O'Reilly Media, 2016, ISBN: 978-1-491-97297-7.

[17] *Netflix Open Source Software Center*, Accessed 7.11.2017. [Online]. Available: https://netflix.github.io/.

[18] A. Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained", Tech. Rep., Accessed 15.5.2018. [Online]. Available: http://www.rgoarchitects.com/Files/fallacies.pdf.

[19] *CircuitBreaker*, Accessed 8.11.2017. [Online]. Available: https://martinfowler.com/bliki/CircuitBreaker.html.

[20] M. Richards, *Microservices AntiPatterns and Pitfalls*. Sebastopol, CA: O'Reilly Media, 2016, ISBN: 978-1-491-96331-9.

[21] *Press Release*, Accessed 26.12.2017. [Online]. Available: https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html.

[22] *Node.js*, Accessed 14.12.2017. [Online]. Available: https://nodejs.org/en/.

[23] *Chrome V8 | Google Developers*, Accessed 14.12.2017. [Online]. Available: https://developers.google.com/v8/.

[24] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs", *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010, ISSN: 1089-7801. DOI: 10.1109/MIC.2010.145.

[25] *The Node.js Event Loop, Timers, and process.nextTick() | Node.js*, Accessed 15.12.2017. [Online]. Available: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/.

[26] *npm*, Accessed 28.12.2017. [Online]. Available: https://www.npmjs.com/.

[27] *GitHub Octoverse 2017 | Highlights from the last twelve months*, Accessed 28.12.2017. [Online]. Available: https://octoverse.github.com/.

[28] *Stack Overflow Developer Survey 2017*, Accessed 28.12.2017. [Online]. Available: https://insights.stackoverflow.com/survey/2017.

[29] *Cloud Application Platform | Heroku*, Accessed 9.1.2018. [Online]. Available: https://www.heroku.com/home.

[30] *Salesforce.com Buys Heroku For 212 Million In Cash | TechCrunch*, Accessed 9.1.2018. [Online]. Available: https://techcrunch.com/2010/12/08/breaking-salesforce-buys-heroku-for-212-million-in-cash/.

[31] *Amazon Web Services (AWS) - Cloud Computing Services*, Accessed 10.1.2018. [Online]. Available: `https://aws.amazon.com/`.

[32] *Amazon EC2*, Accessed 10.1.2018. [Online]. Available: `https://aws.amazon.com/ec2/`.

[33] *The Twelve-Factor app*, Accessed 10.1.2018. [Online]. Available: `https://12factor.net/`.

[34] *Development and Configuration Principles | Heroku Dev Center*, Accessed 25.1.2018. [Online]. Available: `https://devcenter.heroku.com/articles/development-configuration`.

[35] *Leiningen*, Accessed 25.1.2018. [Online]. Available: `https://leiningen.org/`.

[36] *pip - pip 9.0.1 documentation*, Accessed 25.1.2018. [Online]. Available: `https://pip.pypa.io/en/stable/`.

[37] *Runtime Principles | Heroku Dev Center*, Accessed 25.1.2018. [Online]. Available: `https://devcenter.heroku.com/articles/runtime-principles`.

[38] *Jetty - Servlet Engine and Http Server*, Accessed 25.1.2018. [Online]. Available: `https://www.eclipse.org/jetty/`.

[39] *Thin - yet another web server*, Accessed 25.1.2018. [Online]. Available: `http://code.macournoyer.com/thin/`.

[40] *The Process Model | Heroku Dev Center*, Accessed 25.1.2018. [Online]. Available: `https://devcenter.heroku.com/articles/process-model`.

[41] *Dynos and the Dyno Manager | Heroku Dev Center*, Accessed 25.1.2018. [Online]. Available: `https://devcenter.heroku.com/articles/dynos`.

[42] *Docker - Build, Ship and Run Any App, Anywhere*, Accessed 26.1.2018. [Online]. Available: `https://www.docker.com/`.

[43] *Container Registry and Runtime (Docker Deploys) | Heroku Dev Center*, Accessed 26.1.2018. [Online]. Available: `https://devcenter.heroku.com/articles/container-registry-and-runtime`.

[44] *Slug Compiler | Heroku Dev Center*, Accessed 26.1.2018. [Online]. Available: `https://devcenter.heroku.com/articles/slug-compiler`.

[45] *Buildpacks | Heroku Dev Center*, Accessed 26.1.2018. [Online]. Available: `https://devcenter.heroku.com/articles/buildpacks`.

[46] *The leading operating system for PCs, IoT devices, servers and the cloud | Ubuntu*, Accessed 26.1.2018. [Online]. Available: `https://www.ubuntu.com/`.

[47] *HTTP Routing | Heroku Dev Center*, Accessed 26.1.2018. [Online]. Available: `https://devcenter.heroku.com/articles/http-routing`.

[48] *Add-ons - Heroku Elements*, Accessed 26.1.2018. [Online]. Available: https://elements.heroku.com/addons.

[49] *Local and Remote Notification Programming Guide: APNs Overview*, Accessed 22.3.2018. [Online]. Available: https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/APNSOverview.html.

[50] *Cloud Messaging | Google Developers*, Accessed 22.3.2018. [Online]. Available: https://developers.google.com/cloud-messaging/.

[51] *Windows Push Notification Services (WNS) overview - UWP app developer | Microsoft Docs*, Accessed 22.3.2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/uwp/design/shell/tiles-and-notifications/windows-push-notification-services--wns--overview.

[52] *Cluster | Node.js v8.11.0 Documentation*, Accessed 28.3.2018. [Online]. Available: https://nodejs.org/docs/latest-v8.x/api/cluster.html.

[53] *Child Process | Node.js v8.11.0 Documentation*, Accessed 29.3.2018. [Online]. Available: https://nodejs.org/docs/latest-v8.x/api/child_process.html.

[54] *throng - npm*, Accessed 29.3.2018. [Online]. Available: https://www.npmjs.com/package/throng.

[55] *Optimizing Dyno Usage | Heroku Dev Center*, Accessed 29.3.2018. [Online]. Available: https://devcenter.heroku.com/articles/optimizing-dyno-usage.

[56] *Express - Node.js web application framework*, Accessed on 29.3.2018. [Online]. Available: https://expressjs.com/.

[57] *Pinqvin/masters-thesis-example-backend: Example Node.js backend for my masters thesis to demonstrate clustering*, Accessed 19.4.2018. [Online]. Available: https://github.com/Pinqvin/masters-thesis-example-backend.

[58] *TypeScript - JavaScript that scales*. Accessed 19.4.2018. [Online]. Available: https://www.typescriptlang.org/.

[59] *PostgreSQL - The world's most advanced open source database*, Accessed 19.4.2018. [Online]. Available: https://www.postgresql.org/.

[60] *Gatling Load and Performance testing - Open-source load and performance testing*, Accessed 19.4.2018. [Online]. Available: https://gatling.io/.

[61] *Optimizing Node.js Application Concurrency | Heroku Dev Center*, Accessed 26.4.2018. [Online]. Available: https://devcenter.heroku.com/articles/node-concurrency.

[62] *Scaling Your Dyno Formation | Heroku Dev Center*, Accessed 26.4.2018. [Online]. Available: https://devcenter.heroku.com/articles/scaling.

A. EXAMPLE SERVICE CONTROLLER IMPLEMENTATION

```
import { Request, Response } from 'express';
import joi from 'joi';
import * as DataCore from '../core/data';
import { DataId, dataIdParameter } from '../types/data';
import { validateParameters } from '../util/common';

export async function getDataById(req: Request, res: Response) {
  const id = validateParameters<DataId>(req.params.id, dataIdParameter);

  const result = await DataCore.getDataById(id);

  if (result === undefined) {
    return res.sendStatus(404);
  }

  return res.json(result);
}

export async function getDataByIdCpu(req: Request, res: Response) {
  const id = validateParameters<DataId>(req.params.id, dataIdParameter);

  for (let i = 0; i < 5e7; ++i);
  const result = await DataCore.getDataById(id);

  if (result === undefined) {
    return res.sendStatus(404);
  }

  return res.json(result);
}
```

B. EXAMPLE SERVICE CLUSTERING IMPLEMENTATION

```
import sourceMapSupport from 'source-map-support';
sourceMapSupport.install();

import dotenv from 'dotenv';
dotenv.config();
import { Server } from 'http';
import throng from 'throng';
import createApp from './app';

import logger from './util/logger';
import connect from './db';
import { PORT, WEB_CONCURRENCY } from './config';

function closeServer(server: Server) {
  logger.info('Stopping incoming connections');
  server.close(() => {
    logger.info('Server has stopped listening to incoming connections');
    logger.info('Closing database connection');
    connect().destroy();
  });
}

function startServer(id: number) {
  logger.info('Starting server...');

  const app = createApp();
  const server = app.listen(PORT, () => {
    logger.info(`Server worker ${id} listening on port ${PORT}`);
  });

  process.on('SIGINT', () => closeServer(server));
  process.on('SIGTERM', () => closeServer(server));
}

if (require.main === module) {
  throng({
    workers: WEB_CONCURRENCY,
    start: startServer,
  });
}
```

C. BACKEND GATLING TEST FOR NON-CPU INTENSIVE WORKLOAD

```
1  package data
2
3  import io.gatling.core.Predef._
4  import io.gatling.http.Predef._
5  import scala.concurrent.duration._
6
7  class TestSimulation extends Simulation {
8
9    val httpConf = http
10     .baseURL("http://localhost:3000")
11     .acceptHeader("application/json")
12     .acceptEncodingHeader("gzip, deflate")
13     .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:16.0)")
14
15   val rand = new scala.util.Random()
16
17   val scn = scenario("Scenario Name")
18     .exec(http("request_data")
19       .get(s"/api/data/${rand.nextInt(1000000)}")
20       .check(status.is(200)))
21
22   setUp(scn.inject(rampUsersPerSec(10) to (80) during (30 seconds))
23     .protocols(httpConf))
24 }
```

D. BACKEND GATLING TEST FOR CPU INTENSIVE WORKLOAD

```
1  package data
2
3  import io.gatling.core.Predef._
4  import io.gatling.http.Predef._
5  import scala.concurrent.duration._
6
7  class TestSimulationCpu extends Simulation {
8
9    val httpConf = http
10     .baseURL("http://localhost:3000")
11     .acceptHeader("application/json")
12     .acceptEncodingHeader("gzip, deflate")
13     .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:16.0)")
14
15   val rand = new scala.util.Random()
16
17   val scn = scenario("Scenario Name")
18     .exec(http("request_data")
19       .get(s"/api/data-cpu/${rand.nextInt(1000000)}")
20       .check(status.is(200)))
21
22   setUp(scn.inject(rampUsersPerSec(10) to (80) during (30 seconds))
23     .protocols(httpConf))
24 }
```